RICE UNIVERSITY

A Refined Parallel Simulation of Crossflow Membrane Filtration

by

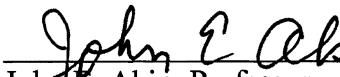
Paul Martin Boyle

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

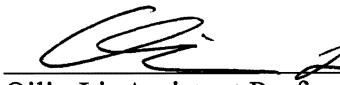
Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Brent C. Houchens, Assistant Professor, Chair
Mechanical Engineering & Materials Science



John E. Akin, Professor
Mechanical Engineering & Materials Science



Qilin Li, Assistant Professor
Civil and Environmental Engineering

HOUSTON, TX

MARCH, 2011

ABSTRACT

A Refined Parallel Simulation of Crossflow Membrane Filtration

By

Paul Martin Boyle

This work builds upon the previous research carried out in the development of a simulation that incorporated a dynamically-updating velocity profile and electric interactions between particles with a Force Bias Monte Carlo method. Surface roughness of the membranes is added to this work, by fixing particles to the membrane surface. Additionally, the previous electric interactions are verified through the addition of an all-range solution to the calculation of the electrostatic double layer potential between two particles. Numerous numerical refinements are made to the simulation in order to ensure accuracy and confirm that previous results using single-precision variables are accurate when compared to double-precision work. Finally, the method by which the particles move within a Monte Carlo step was altered in order to implement a different data handling structure for the parallel environment. This new data handling structure greatly reduces the runtime while providing a more realistic movement scheme for the particles. Additionally, this data handling scheme offers the possibility of using a variety of n-body algorithms that could, in the future, improve the speed of the simulation in cases with very high particle counts.

Acknowledgements

Dr. Brent C. Houchens, my thesis advisor and friend, is owed great thanks for his support, guidance, and insight, without which I wouldn't have been able to complete this work. Many thanks to Dr. Albert Kim, who wrote the original code and carried out the initial research that all of my research is built upon. Finally, thanks to my family, friends, and my wonderful wife, Michelle, who have supported and encouraged me through this process.

Thanks are due for the many hours of use of Rice's Ada cluster, which is run by the Rice Computational Research Cluster and funded by NSF under Grant CNS-0421109, and a partnership between Rice University, AMD and Cray. Additional thanks for the hours on STIC, which is part of the Cyberinfrastructure for Computational Research funded by NSF under Grant CNS-0821727.

Financial support that made this research possible came through the Department of Defense (DoD) National Defense Science and Engineering Graduate (NDSEG) Fellowship, administered by the American Society for Engineering Education (ASEE). Additional support came from the National Science Foundation (NSF) Graduate Research Fellowship Program (GRFP) and Rice University's Nettie S. Autrey Fellowship program.

Table of Contents

List of Figures

List of Tables

Glossary

a	Sphere Radius
a_0	Move Size Coefficient
A_H	Hamaker Coefficient
C	Local Concentration
C_{EL}	Electrolyte Concentration
D	Diffusivity
$\hat{D}$	Shear-Induced Diffusivity Correction Factor
e	Electron Charge
E	Particle Energy
$\mathbf{F}$	Force Vector
h	Scaled Surface to Surface Distance
k_b	Boltzmann Constant
K	Sedimentation Coefficient
l	Shear Plane to Shear Plane Distance
l_L, l_W	Scaled Membrane Surface Dimensions
N_A	Avogadro's Number
NP	Number of Particles
N_s	Number of Surface Particles
P	Pressure
P_i	Particle Total Potential
P_x	Movement Bias in the Crossflow Direction
P_z	Probability a Monte Carlo Move Will be Accepted
R	Particle Radius
R_a	Surface Roughness
$rand$	Random Double Precision Number Between 0.0 and 1.0
Re	Reynolds Number
s	Scaled Center to Center Distance
S	Osmotic Compressibility Coefficient in Brownian Diffusivity
T	Absolute Temperature
v	Velocity
v_{max}	Maximum Velocity in the Crossflow (Original Code)
v_w	Permeate Velocity
$\overline{v}_x$	Mean Crossflow Velocity
V	Electric Potential
x	Crossflow Direction
y	No Flow Direction (Perpendicular to x and z Directions)
z	Center to Center Distance (In van der Waals Potential), Direction Perpendicular to Wall Otherwise
z_p	Valence
$\hat{\mathbf{z}}$	Unit Vector Perpendicular to Cell Wall
Z	Osmotic Compressibility

Greek Symbols

α	Viscosity Exponential Coefficient
γ	Reduced Surface Potential
$\dot{\gamma}$	Shear-Rate
$\Delta\_$	Move Size in x, y, or z Direction, or Change in Energy E
$\mathbf{\Delta r}$	Trial Particle Move Vector
ε_0	Permittivity of Free Space
ε_r	Relative Dielectric Constant
ζ	Zeta Potential
κ	Inverse Debye Screening Length
η	Generalized Newtonian Viscosity
η_0	Pure Newtonain Fluid Viscosity
λ	Force Bias Monte Carlo Coefficient
σ	Normalized Channel Position Between -1.0 and 1.0
ϕ	Suspension Volume Fraction
μ	Fluid Medium Viscosity
μ_m	Mean Surface Height
Ψ_{NP}	Order Parameter

Subscripts

B	Brownian
EDL	Electrostatic Double Layer
PP	Plane-Plane Interaction
S	Surface
SI	Shear-Induced
SS	Sphere-Sphere Interaction
SP	Sphere-Plane Interaction
VDW	Van der Waals
x	Crossflow Direction
z	Direction Perpendicular to Cell Wall

Chapter 1: Background

1.1 Motivation

Access to clean drinking water is a critical issue in the world today. In 2004, over a billion people lacked access to clean drinking water [1]. In 2008, this figure had improved to 884 million people [2]. While this demonstrates improvement, there remains a long way to go to provide everyone with safe drinking water. One of the technologies that shows great promise in helping to address the need for clean drinking water is reverse osmosis membrane filtration. In this process "dirty" water is pushed against its osmotic gradient, generating more concentrated dirty water, known as concentrate, and clean water, known as permeate. This process not only makes the water more visually appealing by removing debris and dissolved solids from the water, it also removes some of the most harmful contaminants from the water, including heavy metals, viruses, and bacteria [3].

With such fine filtration comes a price. The filters are susceptible to fouling where layers of contaminants build up on the membrane surface, resulting in decreased permeate flow and the need to either clean or replace the filters. As the fouling increases, higher pressures are required to push permeate through the membrane, meaning increased power consumption. Membrane fouling is a recognized issue and has prompted substantial study in the last few decades. This is due to the ever increasing presence of membrane filtration in water purification and the decrease in price that has, and will continue to, accompany technological advancements in filtration [4]. A variety of fouling mechanisms, including biofouling [5], colloidal fouling [6], and synergistic fouling mechanisms in which multiple foulants act together at a greater than additive rate to foul

membranes [7] have been identified. The need to overcome these fouling mechanisms has resulted in a variety of experimental simulations [8-10] of a common membrane preserving technique known as crossflow filtration. Crossflow filtration involves the use of pressure driven flow perpendicular to the permeate flow, which sweeps away the concentrate. To better understand this system previous work has been completed to study the link between variable shear rates and viscosity in the crossflow with membrane fouling [11-12].

In addition to understanding the hydrodynamic effects in the membrane system there has been a large amount of work to understand the electric potentials at work in the system [13-14]. These studies look at the effects that the surface charge and charge of the particles being deposited on the surface have on membrane fouling. Methods to decrease or eliminate this fouling have been proposed and tested for many years [15-16]. However, without first understanding the true mechanism behind the fouling it is difficult to propose an effective solution [17]. This work aims develop a better understanding of colloidal fouling of membranes using crossflow in order to mitigate fouling and maximize clean drinking-water production.

1.2 Simulation Overview

This work continues the efforts toward an accurate simulation of membrane filtration utilizing a Force Biased Monte Carlo method. This research is based on a simulation initially developed by Dr. Albert Kim, at the University of Hawaii at Manoa, who assumed no electric interaction in the channel and a constant velocity profile throughout the simulation [18]. A dynamically updating velocity profile based on a spectral collocation solution was then added to the simulation, as were particle-particle, and particle-surface interactions [12]. These modifications raised additional questions, which this current work attempts to answer and address. This work can be broken up into four different sections: parallel optimization, numerical refinement, modeling of surface roughness, and refinement of electric interactions.

On the subject of parallel optimization, we are looking not only to improve speed, but to increase the flexibility of the simulation for future work. The previous work made it impossible to implement any form of advanced N-body algorithms due to the way data was handled. The new data passing and handling structure not only improves the runtime of the simulation, but allows for the implementation of potential lumping algorithms like the Fast Multipole Method (FMM) [19]. While not implemented in the current work, these methods could be powerful tools when significantly increasing the scale of the simulation.

The addition of a dynamically updating velocity profile and the electric interactions made the simulation more complex, and resulted in new numerical issues. These included issues related to the computation of the concentration profile, which is

directly coupled to the velocity profile, and issues with the implementation of the electrostatic potential formulas. These numerical issues are addressed is Chapter 4.

While we have previously accounted for the interaction of the particles in the flow with the membrane surface, this work treated the surfaces as perfectly flat plates. In a real world filtration setup is not the case. Reverse osmosis membranes look more like a tightly woven cheese cloth, making them semi-permeable membranes that remove very fine particles from the water while producing the permeate (drinkable water). To accurately simulate the interaction of the particles with the wall (membrane), we must introduce surface roughness.

The final element addressed in this work is the refinement of the electric interactions. All electrostatic formulae come from Dr. Elimelech's text [20], which lists a variety of ways to calculate the electric interaction between the particles depending on their separation, zeta potential, and a several other factors. The previous work made use of a two-range model, where particles were either in close proximity or distant. The simplicity of these formulae made them attractive options, but here they are tested against one of the more robust equations which can calculate the potential for all separation distances. The results of this comparison are discussed in Chapter 6.

Chapter 2: Problem Formulation

This work is carried out to understand the method by which membranes become fouled in typical crossflow filtration setup illustrated below.

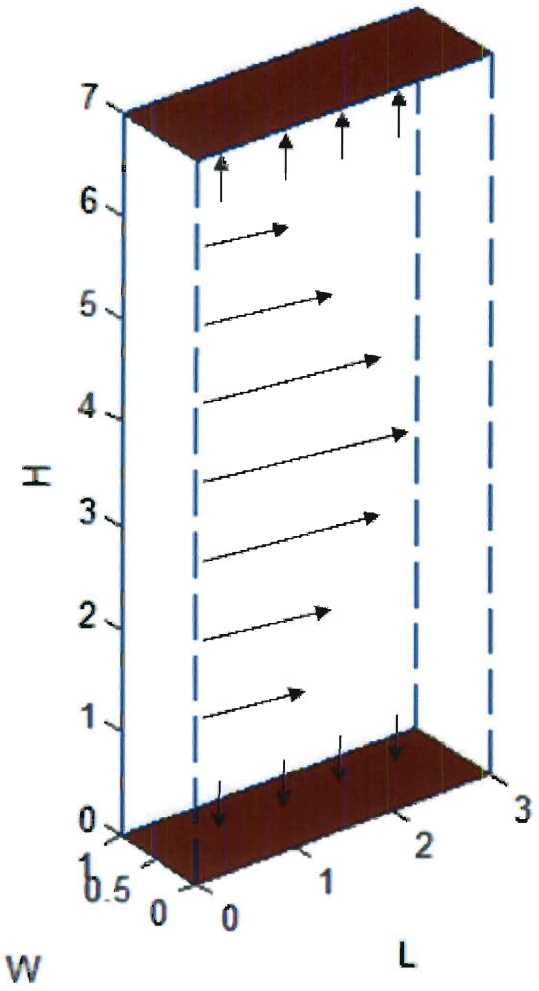


Figure 1: Crossflow Cell

For much of this work, the height-to-length-to-width ratio of the cell used in the simulation is set at 7x3x1. The velocity profile in the crossflow direction (noted as L above) is dynamically updated as the simulation progresses through 10,000 Monte Carlo steps. During each of these steps the particles attempt to move, and the quality of the

move is evaluated, leading to the next state of the simulation. The quality of the move is determined though calculating the energy level of the particle at the new state versus the old state based on electrostatic double layer potentials, van der Waals potentials and velocity of the crossflow at the particle's location in the flow.

2.1 Velocity Profile

The velocity profile in crossflow is governed by the momentum equation in one-dimensional, parallel flow, adapted to use a generalized Newtonian viscosity function [21].

$$0 = -\frac{dP}{dx} + \eta(C)\frac{d^2 v_x}{dz^2} + \frac{d\eta}{dC}\frac{dC}{dz}\frac{dv_x}{dz} \tag{1}$$

In the above equation the function η represents the generalized Newtonian viscosity, which is a function of the particle concentration, C. This function is defined as:

$$\eta(C) = \eta_0 \exp(\alpha C) \tag{2}$$

The constants η_0 and α in the above equation must be determined experimentally and are solution dependent. Several assumptions are made in order to apply this formulation to the problem studied. The flow is assumed to be in quasi-equilibrium in time, the permeate flow in the z direction (noted as H in Figure 1) is negligibly small compared to the crossflow, and there is no flow in the y direction (W above). The boundary conditions imposed on the continuum crossflow profile are no-slip and negligible flow-through at the membrane surfaces (red planes in Figure 1). It should be noted that the permeate flow does affect particle migration, but does not affect the continuum flow. Following this definition the profile is then solved numerically using spectral methods

[22], specifically Chebyshev polynomials with Chebyshev-Gauss-Lobatto collocation points [23].

The velocity profile is highly dependent on the generalized Newtonian viscosity function and therefore on the local concentration in the channel. This concentration is approximated by interpolation about the same collocation points used in the above velocity profile, and represented by cubic splines [24] rather than Chebyshev polynomials [12]. Once the viscosity profile is accurately represented the velocity profile can be calculated, and values that influence the Monte Carlo method (such as shear rate) are readily obtained.

2.2 Electric Interaction

The particles are not only influenced by the flow in the cell, but also by electric interaction with the other particles in the system, as well as interaction with the membrane surfaces. These interactions are modeled through two interactions, the van der Waals (VDW) potential and the electrostatic double layer (EDL) potential. For identical particles these two potentials oppose each other, with the electrostatic double layer negating a portion of the van der Waals potential. When the particles are in very close proximity, or are very close to the membrane surface, the electrostatic double layer potential can overwhelm the van der Waals potential and will cause similarly charged particles to repel each other.

These potentials can be described by a variety of equations [20, 25], and can have different forms for long and short-range interactions. For this work, the short-range van der Waals potential between two spheres of the same size takes the form:

$$V_{VDW-SS} = -\frac{A_{H-SS}}{3}\left[\frac{R^2}{z^2-4R^2}+\frac{R^2}{z^2}+\frac{1}{2}\ln\left(1-\frac{4R^2}{z^2}\right)\right] \quad (3)$$

In this equation, the spheres have radius R and the distance between their centers is given by z. The Hamaker coefficient, A_{H-SS}, depends on the material composition of each sphere and the medium that separates them. A problem to note is that this formula will diverge as the particles come very close to each other. For this reason, a minimum separation between particles must be imposed. When the particles approach closer than this minimum separation they are treated as having overlapped, and such movements are treated as impossible and are rejected. This minimum distance has been noted in several papers [26-27] as roughly 0.158 nm, which is used in this work.

To improve simulation speed a computationally simpler formula for van der Waals potential should be used when possible. These formulae are available for long-range interactions, when the particle separation is much greater than the radius of the particles:

$$V_{VDW-SS} = -A_{H-SS}\frac{16}{9}\frac{R^6}{z^6} \quad (4)$$

For a system of identically sized particles it is simpler to use a normalized system of coordinates, where the unit of distance is a single particle radius. In this system equation (3) becomes:

$$V_{VDW-SS} = -\frac{A_{H-SS}}{3}\left[\frac{1}{s^2-4}+\frac{1}{s^2}+\frac{1}{2}\ln\left(1-\frac{4}{s^2}\right)\right] \quad (5)$$

where the distance $s = z/R$ is referred to as the scaled center-to-center distance between particles. Similarly, equation (4) simplifies as:

$$V_{VDW-SS}(s) = -A_{H-SS}\frac{16}{9}\frac{1}{s^6} \tag{6}$$

The cutoff value above which equation (6) is used in place of equation (5) was previously determined to be approximately $s = 50$ [12].

Like the van der Waals potential, the electrostatic double layer potential has long-range and close-range forms. The general close-range interaction is given by:

$$V_{EDL-SS} = \frac{64\pi a_1 a_2 \varepsilon_0 \varepsilon_r k_b{}^2 T^2}{(a_1 + a_2) z_e{}^2 e^2}\gamma_1\gamma_2 \exp(-\kappa l) \tag{7}$$

Where the Debye screening length is given by:

$$\kappa = \sqrt{\frac{2\times 10^3 N_A e^2 z_e{}^2 C_{EL}}{\varepsilon_0 \varepsilon_r k_b T}} \tag{8}$$

Additionally, the reduced surface potential is described by:

$$\gamma = \tanh\left(\frac{z_p e\zeta}{k_b T}\right) \tag{9}$$

Equation (7) can be simplified for systems of identical particles by examining the newly introduced parameters. The values a_1 and a_2 are the radii of the two particles, which here both equal R. In equation (9) the zeta potentials (ζ) are the same for the two particles, meaning that γ_1 and γ_2 are equal. With this in mind, equation (7) simplifies to:

$$V_{EDL-SS} = \frac{32\pi R \varepsilon_0 \varepsilon_r k_b{}^2 T^2}{z_e{}^2 e^2}\gamma^2 \exp(-\kappa l) \tag{10}$$

For long-range interactions, the general equation has the small change:

$$V_{EDL-SS} = \frac{64\pi a_1 a_2 \varepsilon_0 \varepsilon_r k_b{}^2 T^2}{(a_1 + a_2 + l) z_e{}^2 e^2}\gamma_1\gamma_2 \exp(-\kappa l) \tag{11}$$

This is then simplified similarly for the identical particles case to:

$$V_{EDL-SS} = \frac{64\pi R^2 \varepsilon_0 \varepsilon_r k_b{}^2 T^2}{(2R+l) z_e{}^2 e^2} \gamma^2 \exp(-\kappa l) \tag{12}$$

where l in the above equations is the separation between the shear planes of the two particles. The shear plane is a very small distance from the solid surface of the particle to the point where zeta potential is calculated [20]. While the exact distance is unknown, literature suggests that it is between the Stern plane and the Gouy plane, with the distance separating the two being κ^{-1}. The shear plane may likely be closer to the Gouy plane than the Stern plane [28]. As the Stern plane is very close to the solid surface, and this distance is much less than the distance separating the Stern and Gouy planes, this work assumes that the shear-plane is roughly κ^{-1} from the particle surface, and these results show insignificant differences from cases where the shear plane is neglected.

By examination it can be seen that equation (10) is a special case of equation (12), where l is taken to be very small compared to R. With this in mind, the cutoff at which the transition from close-range to long-range formulas occurs was set to $l = 0.1 \times R$. It is important to note that potential equations (10) and (12) have formal validity regions that are given by:

$$\begin{aligned} l &\ll a_i \\ \kappa a_i &> 5 \end{aligned} \tag{13}$$

and:

$$\begin{aligned} \kappa(2R+l) &\gg 1 \\ \kappa a_i &\geq 10 \end{aligned} \tag{14}$$

The equations can typically be used beyond these ranges [26-27, 29], but here the results of a simulation using these equations with the results of a simulation using equations with alternative validity ranges are compared (as discussed in Chapter 6).

In addition to the particle-particle interactions, there are particle-surface interactions to consider. For the van der Waals potential this is a special case of equation (5) where one particle radius is allowed to go to infinity:

$$V_{VDW-SP} = -\frac{A_{H-SP}}{6}\left[\frac{R}{l} + \frac{R}{2R+l} + \ln\left(\frac{l}{2R+l}\right)\right] \tag{15}$$

This form holds for all separation distances between the particle and surface. If another normalized parameter h is defined as $h = l/R$ this equation simplifies to the form:

$$V_{VDW-SP} = -\frac{A_{H-SP}}{6}\left[\frac{1}{h} + \frac{1}{2+h} + \ln\left(\frac{h}{2+h}\right)\right] \tag{16}$$

One important quantity in this equation is the Hamaker coefficient between the particle and the membrane surface. This is calculated using a geometric mean of the Hamaker coefficients of two like substance across water [30]:

$$A_{H-SP} = \sqrt{A_{H-SS}A_{H-PP}} \tag{17}$$

For the electrostatic double layer potential the particle-surface formula is found in similar fashion as the van der Waals formula. Taking the limit of equations (7) or (11) as one particle radius goes to infinity, the formula is found to be:

$$V_{EDL-SP} = \frac{64\pi R\varepsilon_0\varepsilon_r {k_b}^2 T^2}{z_e e^2}\gamma_1\gamma_2 \exp(-\kappa l) \tag{18}$$

The difference between γ_1 and γ_2 is maintained, as the membrane surface zeta potential is not necessarily the same as that of the particles in the simulation.

2.3 Force Bias Monte Carlo Method

Using the velocity profile and electric potentials above, a Monte Carlo method is applied to simulate the motion of the particles. This method is known as the Force Bias Monte

Carlo method, where proposed motions of particles are more likely to be accepted if they move in the direction of the hydrodynamic force [18]. Additionally, this work makes use of a "smart" movement sizing system, where the particles are displaced as:

$$\begin{aligned} \Delta x &= a_0\left(2\times rand - 1 + P_x\right)\left(0.1 + P_x\right) \\ \Delta y &= a_0\left(2\times rand - 1\right)\left(0.1 + P_x\right) \\ \Delta z &= a_0\left(2\times rand - 1\right)\left(0.1 + P_x\right) \end{aligned} \tag{19}$$

In this system rand is a random double precision value between 0.0 and 1.0, and a_0 is a movement sizing parameter that is updated to maintain the movement acceptance ratio around 0.5. The final portion of each displacement, $\left(0.1 + P_x\right)$, helps to scale the moves with respect their position in the crossflow. P_x is defined as:

$$P_x = \frac{v_x(\sigma)}{\max\left(v_x(\sigma)\right)} \tag{20}$$

where σ is the normalized displacement from the center of the cell in the H direction shown in Figure 1, varying from -1 to 1. Assuming a parabolic velocity profile, the possible displacements in the crossflow direction are illustrated below.

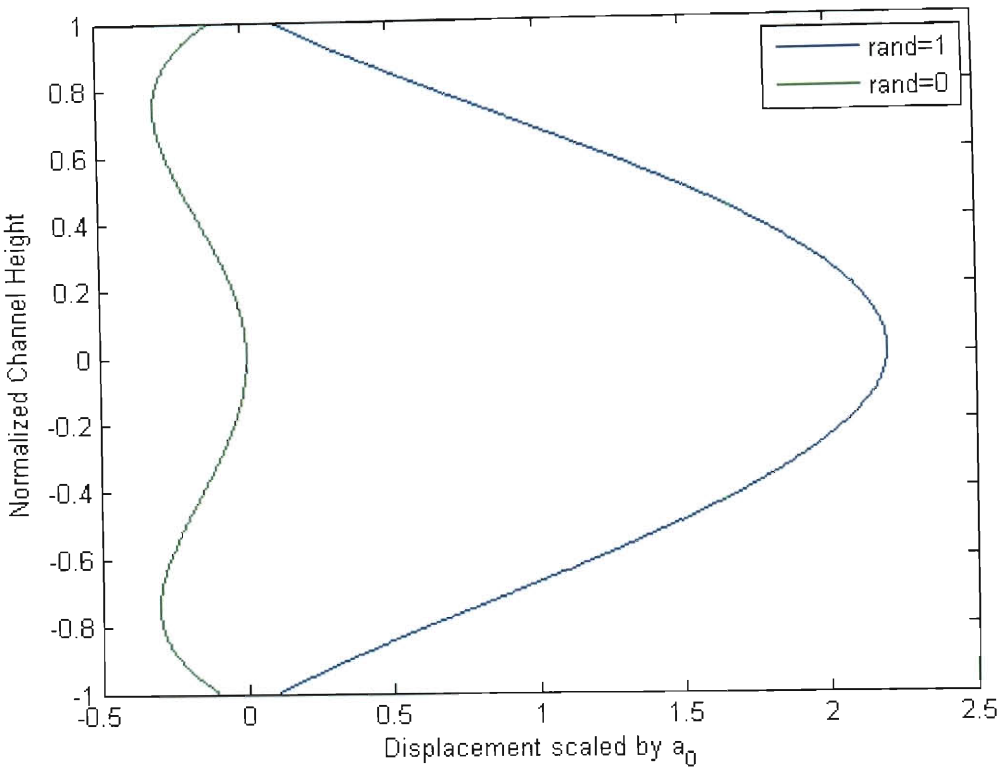


Figure 2: Possible Displacements in the Crossflow Direction for a Parabolic Velocity Profile

The probability of movement acceptance is evaluated based on the formula:

$$P_z = \min\left[1, \exp\left(-\beta\Delta E - \beta\lambda \mathbf{F}_h \cdot \mathbf{\Delta r}\right)\right] \tag{21}$$

This function allows that if the particle is moving to a lower energy state the move will always be accepted. Otherwise the probability that a move to a higher energy state will be accepted is calculated. The values β and λ are constants in the simulation, with $\beta = 1/k_b T$, and λ being the force bias coefficient set to 0.5 [31-32]. The hydrodynamic force, $\mathbf{F}_h$, is a function of the Brownian and shear-induced diffusivities, as well as the velocity of the permeate flow at that point in the cell:

$$\mathbf{F}_h = \frac{k_b T v_z(\sigma) K^{-1}(\phi)}{D_B + D_{SI}} \hat{\mathbf{z}} \tag{22}$$

where v_z is the permeate flow velocity, D_B is the Brownian diffusivity, D_{SI} is the shear-induced diffusivity, and K is the sedimentation coefficient which is a function of total particle volume fraction ϕ.

The permeate velocity as a function of the normalized channel height is described by Berman [33] as:

$$v_z(\sigma) = v_w\left[\frac{\sigma}{2}\left(3-\sigma^2\right)-\frac{Re}{280}\sigma\left(2-3\sigma^2+\sigma^6\right)\right] \tag{23}$$

where v_w is the permeate velocity through the membrane surface. Assuming that the Reynolds number of the permeate flow (Re) is negligibly small this formula simplifies to the form:

$$v_z(\sigma) \simeq v_w\frac{1}{2}\sigma\left(3-\sigma^2\right) \tag{24}$$

This velocity profile is used with Happel's sphere-in-cell model [34] to form the numerator of the hydrodynamic force. This requires the definition of the sedimentation coefficient K, which Happel approximates the inverse of as:

$$K^{-1}(\phi) = \frac{6+4\phi^{5/3}}{6-9\phi^{1/3}+9\phi^{5/3}-6\phi^2} \tag{25}$$

The Brownian diffusivity can be described as:

$$D_B = \frac{k_b T}{6\pi\mu R}S(\phi) \tag{26}$$

where S is a function of the osmotic compressibility, Z:

$$S(\phi) = \frac{\partial\phi Z(\phi)}{\partial\phi} \tag{27}$$

For hard-sphere systems as in this simulation, Z is given by the Carnahan-Starling equation [35]:

$$Z(\phi) = \frac{1+\phi+\phi^2-\phi^3}{(1-\phi)^3} \tag{28}$$

For small total volume fractions S can be approximated as unity [35], and will be treated as such in this work.

The shear-induced diffusivity is dependent on the shear rate in the crossflow, which is defined as:

$$\dot{\gamma}(\sigma) = \frac{dv_x}{d\sigma} \tag{29}$$

This is then used in the diffusivity formula:

$$D_{SI} = \dot{\gamma}(\sigma) R^2 \hat{D}(\phi) \tag{30}$$

where the correction factor $\hat{D}$ has been determined experimentally [36-38] to follow the function:

$$\hat{D}(\phi) = \frac{1}{3}\phi^2\left(1+\frac{1}{2}e^{8.8\phi}\right) \tag{31}$$

With all of these values determined, the probability function in equation (21) is evaluated using the proposed movement vector $\mathbf{\Delta r}$. This probability is compared against a random value between 0.0 and 1.0 to determine if the movement should be accepted.

During the simulation a variety of parameters are recorded at regular intervals to assess the current state and the progress towards convergence to a steady state. These values include the movement acceptance ratio (the percentage of particle movements accepted in each Motne Carlo step), the movement size coefficient a_0, the maximum velocity in the channel, and a parameter that measures the clustering of the particles near

the membrane surface and their symmetry in the cell. This final parameter is known as the order parameter, and is defined as [18]:

$$\Psi_{NP} = \left(\frac{1}{NP} \sum_{i=1}^{NP} \sigma^2 \right) - \left(\frac{1}{NP} \sum_{i=1}^{NP} \sigma \right)^2 \tag{32}$$

where *NP* is the number of particles in the simulation. As particles become more clustered toward the membrane surface, the first term in this equation approaches 1. The second term trends toward zero in a highly symmetric system. This provides an easy to understand measure of the state of the system.

Chapter 3: Parallel Optimization and Benchmarking

Nearly all simulations for the purposes of parallel optimization were carried out on Rice University's Ada cluster, which is a 64-bit Cray XD1 cluster, with 632 dual core AMD processors. The simulations are all coded in FORTRAN90, and utilize the MPI 1.2.6 library for purposes of running the code in parallel. When running in parallel in sections 3.1 - 3.3 the job is always executed on 7 cores, distributed across 7-nodes with a maximum allowable runtime of 8 hours.

3.1 Previous Parallel Implementation

The simulation consists of an inner and an outer loop, the inner loop known as the particle loop, and the outer being the Monte Carlo loop. In the previous implementation (both serial S1 and parallel P1), during each Monte Carlo iteration the particles were moved sequentially and tested to determine if the move was acceptable according to the probability function described in equation (21). After one particle moved and was tested, the next particle would follow the same routine. This resulted in the first particle to move potentially seeing a very different electric state than the last particle to move inside of a single Monte Carlo iteration.

On top of this being physically unsatisfying, this setup was also inefficient on a parallel machine in its initial implementation (P1). This setup required all of the processes to synchronize upwards of four times per particle movement, which is then multiplied by the number of Monte Carlo iterations. Each of these synchronizations increased the runtime and slowed the simulation.

This mode of moving the particles and testing the probability of move acceptability was preserved from the serial form of the simulation (S1) when porting the

code for parallel use (P1). When the simulation was run as a single process on Ada in serial form, the runtime was approximately seven hours. To decrease the runtime of the simulation the computation of electric interactions between the current particle of interest in the Monte Carlo simulation was divided up among seven processes. When this division of work was implemented the performance for the P1 code improved to a runtime of four hours and twenty-eight minutes, with the parameters as defined in our previous work [12] (improved parameter values are used in future chapters). The flow of the code is shown below for reference purposes.

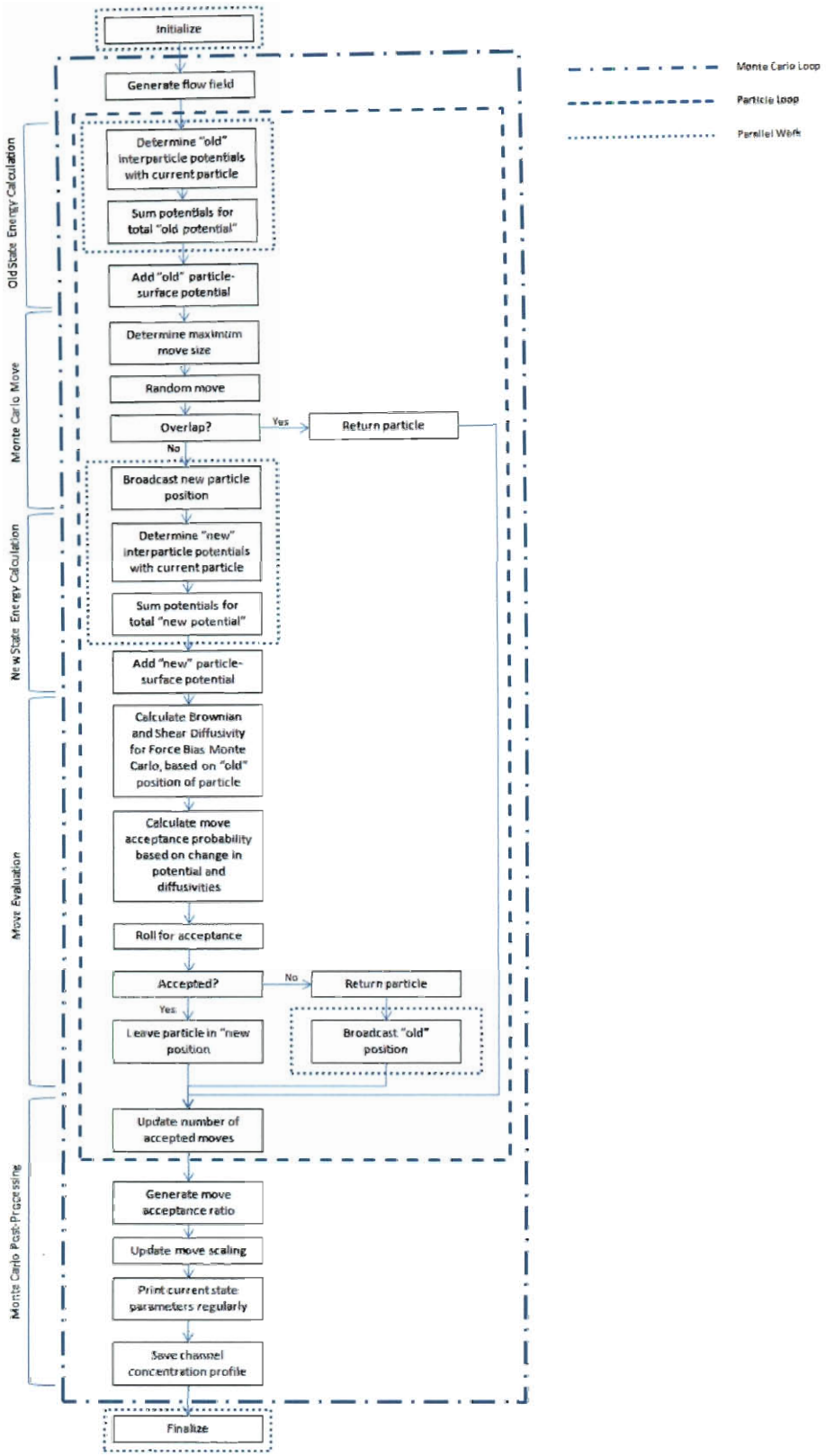


Figure 3: Flow of Previous S1 and P1 Simulation Codes

3.2 New Parallel Implementation

As mentioned previously, the serial code (S1) and initial parallel code (P1) moved particles in a systematic fashion, where the particle moving saw a different state of the simulation than the previous particle. Physically this concept is unsettling. Particles do not move in such an orderly fashion, in reality they all move simultaneously and would see a continuous evolution of the state of the system. The new parallel implementation (P2) attempts to make simulation moves in a more realistic fashion, while improving the efficiency of the simulation on a parallel architecture. Statistically, there should be no significant physical difference between the final steady state results of the P1 and P2 implementations, due to the stochastic nature of the Monte Carlo simulation. The new P2 implementation does however provide better parallel optimization possibilities. In the transient region, particles migrate in a similar way, in a similar number of MC steps, in both the P1 and P2 implementations.

Upon inspecting the code in the P1 implementation, it was noted that there was room for improvement allowing for speedup of the parallel portions. The previous implementation utilized the passing of small packets of data many times per Monte Carlo iteration. The largest of these packets contained the three dimensional coordinate location of the current particle of interest, either in its previous state or in its new position. This required a substantial number of synchronizations of the processes per simulation, in the worst case (where all 2100 particles in each step propose a valid movement without overlap) there would be upwards of 84,000,000 occassions at which the processes would all have to be at the same point. In contrast, when the data handling

structure was changed, there were only 4 synchronizations required per Monte Carlo step, which means 40,000 occassions when the processes had to align.

The tradeoff was that instead of passing small packets of only three double precision variables at once, the largest piece of data that needed to be passed to all processes was a vector of 2,206,050 double precision values. These values are equally distributed across all of the processes in the simulation and must be passed from their local process to all processes in the system. Fortunately, MPI does not require all of the values to be gathered to a single process and then passed to all. We bypass this bottleneck by utilizing MPI's ALLGATHER command, which efficiently passes all of these smaller vectors of data between the processes.

When the calculations are carried out at the process level the particle-particle and particle-surface electric interactions are stored in a local vector. Once ALLGATHER is executed these smaller vectors are buffered locally on each process in the form of a rectangular matrix. This matrix is ordered, but not in a fashion easily accessible for determining the total potential state for each particle. Below is an example of this matrix for a 2,100 particle system running on seven processes. At the beginingof the simulation the all of calculations to be performed per Monte Carlo step are assigned paprticle-partilce interaction numbers. These calcuations are divded evenly among the processes and the calculation numbers are used after the intereacions are calculated to reorganize the values into a usable matrix.

$$\begin{bmatrix} E_{1} & E_{314851} & E_{629701} & E_{944551} & E_{1259401} & E_{1574251} & E_{1889101} \\ E_{2} & E_{314852} & E_{629702} & E_{944552} & E_{1259402} & E_{1574252} & E_{1889102} \\ E_{\ldots} & E_{\ldots} & E_{\ldots} & E_{\ldots} & E_{\ldots} & E_{\ldots} & E_{\ldots} \\ E_{314850} & E_{629700} & E_{944550} & E_{1259400} & E_{1574250} & E_{1889100} & E_{2203950} \\ E_{1,S} & E_{301,S} & E_{601,S} & E_{901,S} & E_{1201,S} & E_{1501,S} & E_{1801,S} \\ E_{2,S} & E_{302,S} & E_{602,S} & E_{902,S} & E_{1202,S} & E_{1502,S} & E_{1802,S} \\ E_{\ldots,S} & E_{\ldots,S} & E_{\ldots,S} & E_{\ldots,S} & E_{\ldots,S} & E_{\ldots,S} & E_{\ldots,S} \\ E_{300,S} & E_{600,S} & E_{900,S} & E_{1200,S} & E_{1500,S} & E_{1800,S} & E_{2100,S} \end{bmatrix} \tag{33}$$

Each column represents the output from a different process of particle-particle interactions ($E_{\ldots}$) and the particle-surface interactions ($E_{\ldots,S}$). On each local process the matrix is reparsed and the calculation number decoded into the pair of particles it represents. This forms a square lower triangular matrix of the particle-particle interactions, shown below.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ E_{1,2} & 0 & 0 & 0 & 0 & 0 \\ E_{1,3} & E_{2,3} & 0 & 0 & 0 & 0 \\ E_{1,\ldots} & E_{2,\ldots} & E_{3,\ldots} & 0 & 0 & 0 \\ E_{1,N-1} & E_{2,N-1} & E_{3,N-1} & E_{\ldots,N-1} & 0 & 0 \\ E_{1,N} & E_{2,N} & E_{3,N} & E_{\ldots,N} & E_{N-1,N} & 0 \end{bmatrix} \tag{34}$$

Due to the method in which particles moved in the previous parallel work the symmetry of the electric interactions could not be exploited. However with all particles moving at once in the current work, this symmetry is available. The above matrix in equation (34) is transposed and added to itself to form an $N \times N$ matrix of the electric interaction between all particles.

$$\begin{bmatrix} 0 & E_{1,2} & E_{1,3} & E_{1,\ldots}^{\ T} & E_{1,N-1} & E_{1,N} \\ E_{1,2} & 0 & E_{2,3} & E_{2,\ldots}^{\ T} & E_{2,N-1} & E_{2,N} \\ E_{1,3} & E_{2,3} & 0 & E_{3,\ldots}^{\ T} & E_{3,N-1} & E_{3,N} \\ E_{1,\ldots} & E_{2,\ldots} & E_{3,\ldots} & 0 & E_{\ldots,N-1}^{\ T} & E_{\ldots,N}^{\ T} \\ E_{1,N-1} & E_{2,N-1} & E_{3,N-1} & E_{\ldots,N-1} & 0 & E_{N-1,N} \\ E_{1,N} & E_{2,N} & E_{3,N} & E_{\ldots,N} & E_{N-1,N} & 0 \end{bmatrix} \tag{35}$$

In practice the lower triangular matrix in equation (34) and its transpose are not actually formed, as this would be computationally inefficient. The symmetry principle is instead utilized to formulate the matrix in equation (35) directly. To this matrix we append a column of the particle-surface interactions. With all of these values in the matrix, we can describe the complete potential state for each particle.

$$\begin{bmatrix} 0 & E_{1,2} & E_{1,3} & E_{1,\ldots}^{\ T} & E_{1,N-1} & E_{1,N} & E_{1,S} \\ E_{1,2} & 0 & E_{2,3} & E_{2,\ldots}^{\ T} & E_{2,N-1} & E_{2,N} & E_{2,S} \\ E_{1,3} & E_{2,3} & 0 & E_{3,\ldots}^{\ T} & E_{3,N-1} & E_{3,N} & E_{3,S} \\ E_{1,\ldots} & E_{2,\ldots} & E_{3,\ldots} & 0 & E_{\ldots,N-1}^{\ T} & E_{\ldots,N}^{\ T} & E_{\ldots,S} \\ E_{1,N-1} & E_{2,N-1} & E_{3,N-1} & E_{\ldots,N-1} & 0 & E_{N-1,N} & E_{N-1,S} \\ E_{1,N} & E_{2,N} & E_{3,N} & E_{\ldots,N} & E_{N-1,N} & 0 & E_{N,S} \end{bmatrix} \tag{36}$$

The total potential state of each particle is now found by summing the member of each row in the above matrix. If the surface, S, is treated as the $N+1^{\text{th}}$ particle then sums below can easily be carried out as

$$P_i = \sum_{j=1}^{N+1} E_{i,j} \tag{37}$$

These summations are done in parallel, dividing all of the particles up evenly amongst the processes. After the summations are completed, the data is gathered on the root process and stored as part of the particles individual information. This is carried out twice per Monte Carlo iteration, once for the old state of the system, before the particles have attempted a move, and again after they move. Following the computation of the potential

states, the move is evaluated for acceptance utilizing equation (21). Should a move fail to pass the probability test, the particle is then returned to its "old" position, prior to the movement attempt. This, however, raises a new issue that was not present in the old parallel implementation.

3.3 Particle Return Cascade

With all of the particles moving at once, the possibility arises that a particle may make a bad move but be unable to return to its previous position because a new particle now occupies that space, or at least part of it. Should this occur, what should be done with these particles? Inspecting the previous implementation, we find that if a particle makes a bad move, it would return to its original position and block the new particle from occupying its space. With this is in mind, the obvious solution is to return the particle that made a bad move, as well as returning the particle that occupies its original space. The process is illustrated below.

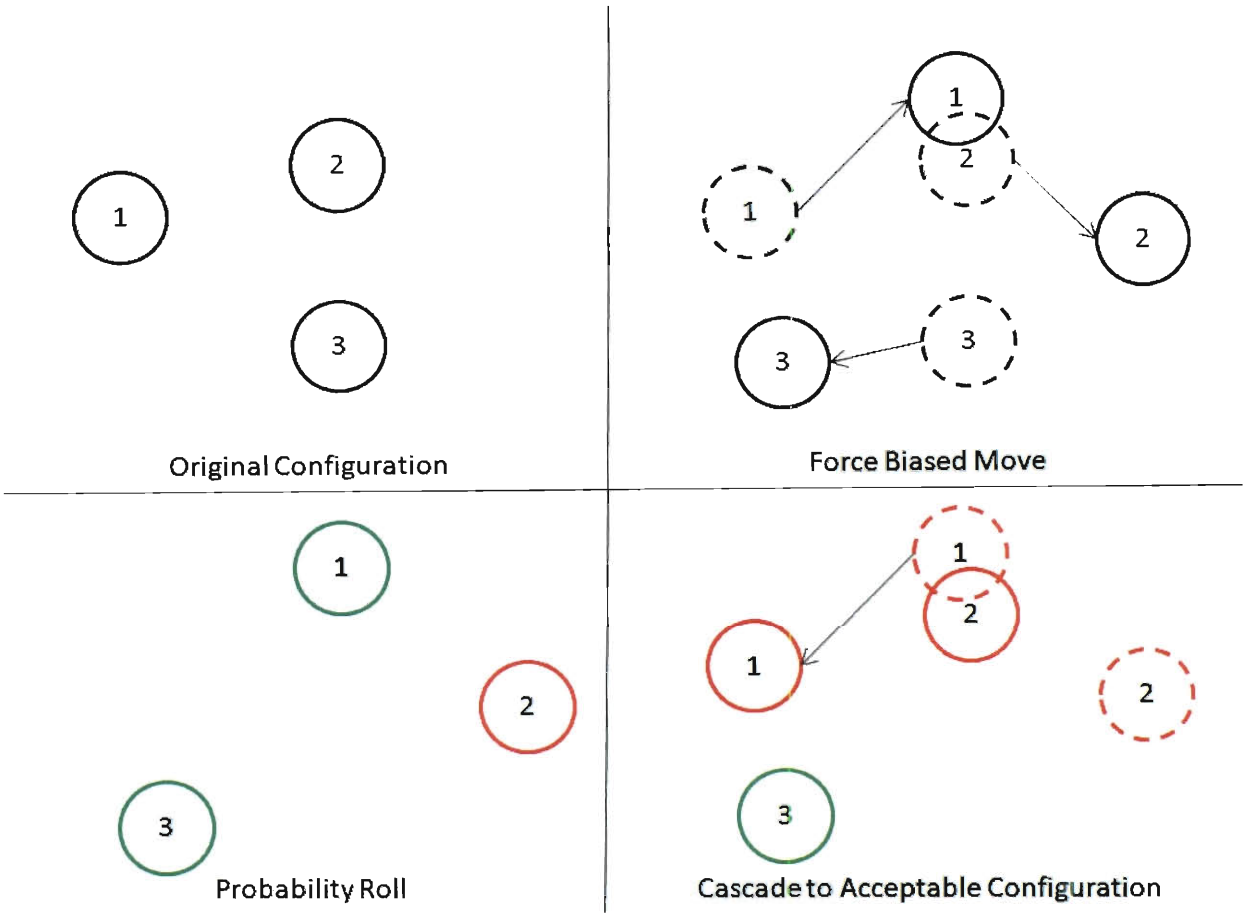


Figure 4: Process for Particle Return Cascade

In the above figure there are four steps to be considered. The original configuration is the state of the particles prior to any proposed moves. In the second step (top-right) the particles are randomly moved taking into account the movement sizing in equation (19). In the third step (bottom-left) the probability of move acceptance is calculated using equation (21) and is compared against a random value from 0 to 1. In this example, the movements of particles 1 and 3 are accepted (green), but the movement of particle 2 is rejected (red). Particle 2 is returned to its original state; however, this position now overlaps with the new position of particle1. Particle 1 is now deemed to have also made a bad move (red), and is returned to its original position as well. This leads to the final

configuration (bottom-right), where particles 1 and 2 are returned to their original position and particle 3 has the only accepted move.

Should particle 1 in Figure 4 encounter a particle occupying its original space, that particle will also be returned. For this reason, the cascade subroutine is applied recursively, checking for particle overlaps after each particle is returned. While this take a substantial amount of time, it remains more efficient than the previous parallel application.

With the above changes, the simulation was repeated to confirm that the same output is obtained regardless of the parallel implementation (P1 or P2). It should be noted that new electric parameters are used throughout this work that correct for previous preliminary numbers used in our previous work [12]. With these new parameters the simulation is not as stable in its final state as before. This instability and oscillation of the output variables means that fewer decimal places are available for the final order parameter and maximum channel velocity. When both methods are executed at a Reynolds number of 1,000 we find the final order parameter (equation (32)) for the old implementation is approximately 0.576, and the maximum velocity in the channel is approximately 1.47 m/s. With the new implementation, the order parameter is 0.592 and the maximum velocity is roughly 1.49 m/s. This is a relatively small change of approximately 2.8% for the order parameter and 1.4% for the maximum channel velocity. At lower Reynolds numbers the difference is even less, for Reynolds number of 0.1, the order parameter is 0.8065 in the P1 model, and 0.8242 for the P2 model, a difference of 2.1%. The maximum channel velocity in the both models was predicted to be 0 m/s out

to the fourth decimal place. These differences can be explained by the change in the way the particles propose and accept their movements.

In terms of efficiency, the new P2 implementation is far superior. The old implementation P1 took approximately 247 minutes for runs of 2,100 particles for 10,000 Monte Carlo steps at a Reynolds number of 1,000. The new data handling implementation (P2) takes only 153 minutes to accomplish the same run, a 37.7% reduction in runtime. At lower Reynolds numbers, for example 0.1, the difference is not as pronounced, with only a 12.6% runtime reduction from 239 to 209 minutes. The flow of the new P2 implementation is graphically described in the chart below.

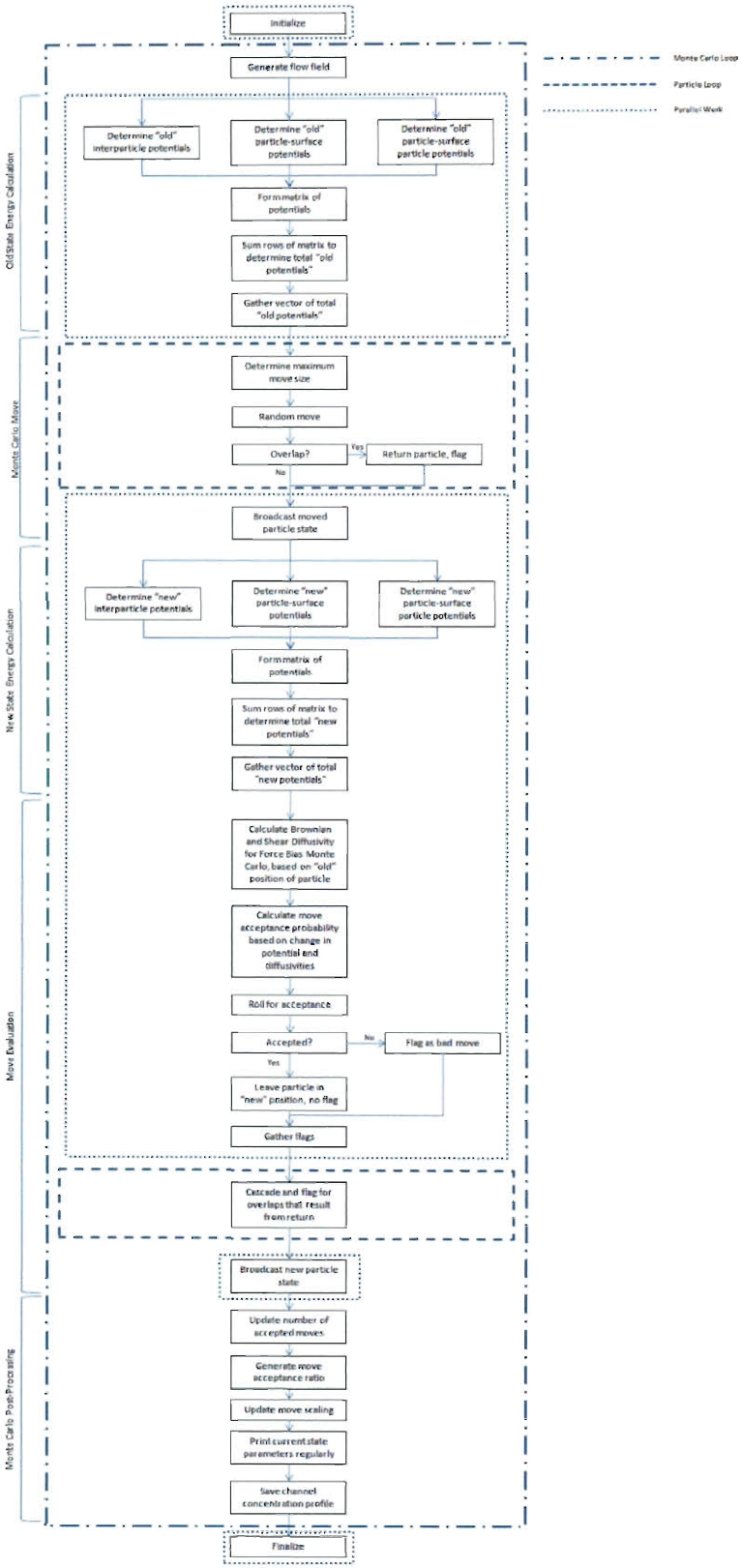


Figure 5: Flow of New P2 Simulation Code

3.4 Increased Simulation Size

One of the primary goals of this work was to increase the efficiency of the simulation in order to handle more particles. All of the previous simulations were run with 2,100 particles on 7 processors. The simulation size was increased by doubleing the number of particles in the system, while maintaining the same number of processors. The original P2 simulation with 2,100 particles took 9,172 seconds to run on 7 processors. For 4,200 particles, 10,000 Monte Carlo steps required 45,225 seconds on 7 processors – more than an NP^2 increase. When the number of processors was doubled to 14, the time required dropped somewhat to 41,533 seconds.

To further expand the number of processors utilized the number of particles was adjusted to better fit with the computer architecture. To do this, the number of particles was increased from 4,200 particles to 5,120 particles and run on 4, 8, 16, 32, and 64 processors. In the 4 processor case the simulation ran up against the maximum runtime limit on the system of 24 hours and terminated after approximately 8,850 Monte Carlo steps. In the remaining cases, an optimum number of processors was found, as shown in the plot below.

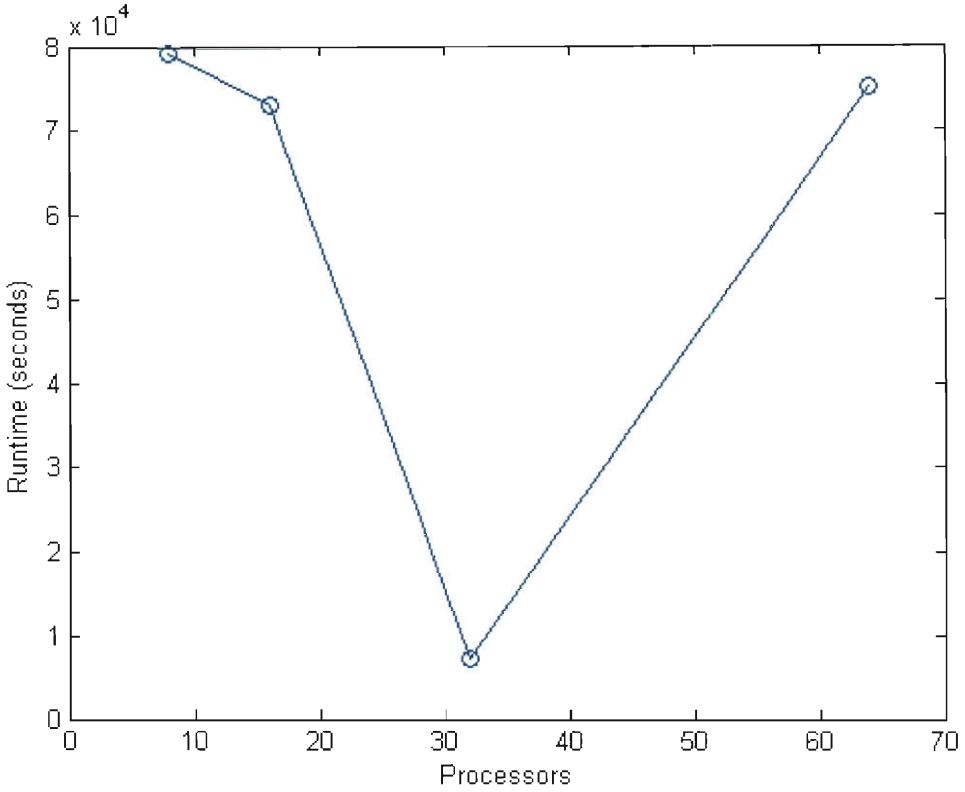


Figure 6: Ada Runtime for 5,120 Particles on Varying Number of Processors at *Re*=1,000

During this work the cluster Ada was decommissioned after suffering a major hardward failure. The simulation was then ported to Rice's STIC supercluster, and a similar study was performed.

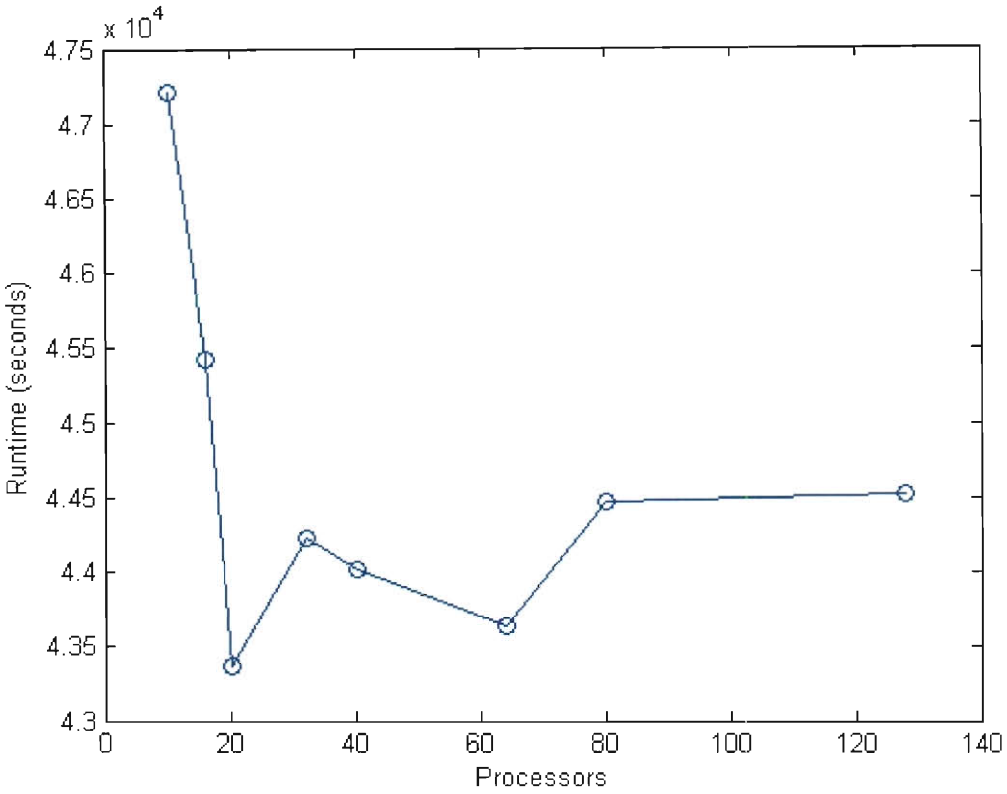


Figure 7: STIC Runtime for 5,120 Particles on Varying Number of Processors at *Re*=1,000

From the above plot it can be seen that there are two local minima in runtime, at 20 and 64 processors. Much of the work after this study was carried out on 64 processors, as the global minima was not discovered until the plot was refined from the original data points of 16, 32, 64, and 128 processors.

The next study was to determine how the simulation scaled with particle number when given a constant number of processors and fixed Reynolds number. The simulation was run on 64 processors at a Reynolds number of 1,000 with 640, 1,280, 2,560, 5,120, and 10,240 particles. While the first four runs were short and fit within STIC's 8 hour runtime window, the last two runs required additional time. The original serial code from Dr. Kim included the ability to restart the simulation from the final state of another

simulation. This routine was updated and adapted for use in STIC's parallel environment allowing for much longer runs.

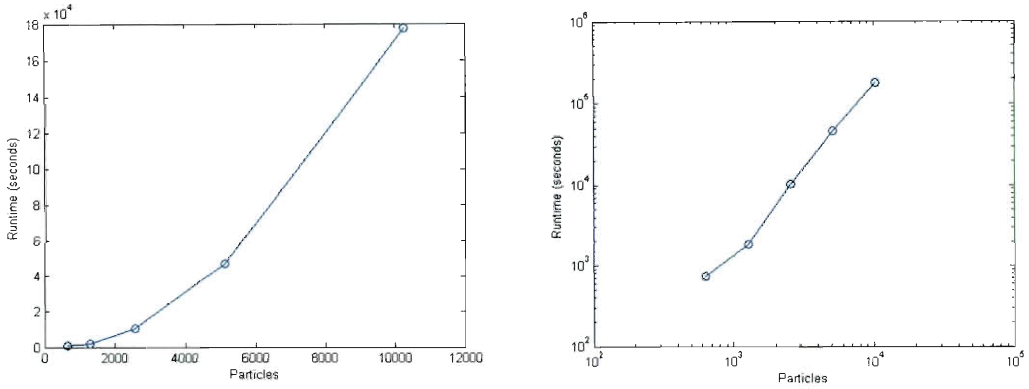


Figure 8: Various Particle Counts for Simulations Run on 64 Processors at *Re*=1,000, on Linear and Log Plots

These plots show roughly quadratic growth of the runtime versus the number of particles in the systems. When the new data handling structure (P2) is compared with the old struture (P1), it is easily observed that at low particle count the new implementation is far superior. For 640 particle run on 20 processors the new P2 implementation completes 10,000 Monte Carlo steps in 666.20 seconds, while the old systems takes more than five times as long at 3646.98 seconds. However, as the size of the simulation increases, a breakeven point is encountered.

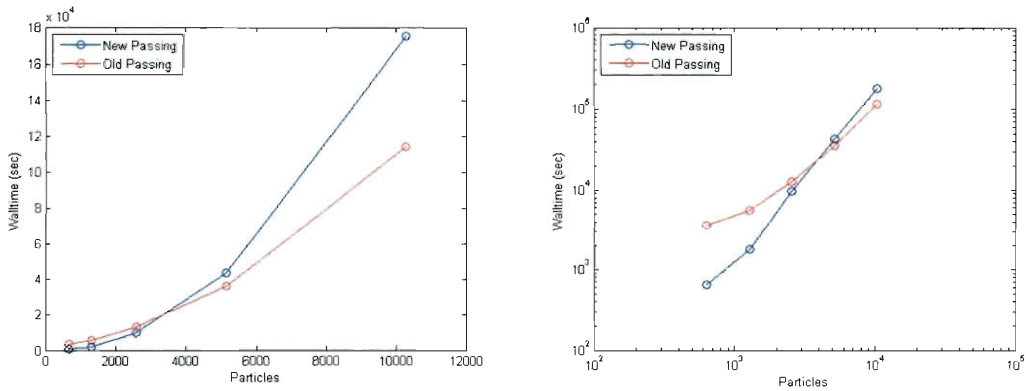


Figure 9: Runtime Comparison between New (P2) and Old (P1) Passing Routines on 20 Processors

As in the 64 processor case, the new P2 data handling system exhibits quadratic growth with the number of particles when run on 20 processors. The old P1 implementation does not exhibit this quite as clearly, and grows at variable rates. While at first the new P2 implementation performs much better, after the particle count increases above 2,560 the old P1 implementation outperforms the new P2 implementation.

If the formation of the matrix of potentials is bypassed in the P2 routine, and the summation of the total potentials is performed by using a direct call to MPI's ALLREDUCE, the simulation both saves on memory usage and speeds up dramatically. The new passing method without the matrix formation (P2.1) takes only 16 hours to complete with 10,240 particles. With the matrix being formed the P2 routine takes three times as long, consuming roughly 48 hours of runtime. Memory usage in the P2.1 implementation for 10,240 particles is approximately 6.4 GB of combined physical and virtual memory, down from 54 GB in the P2 implementation.

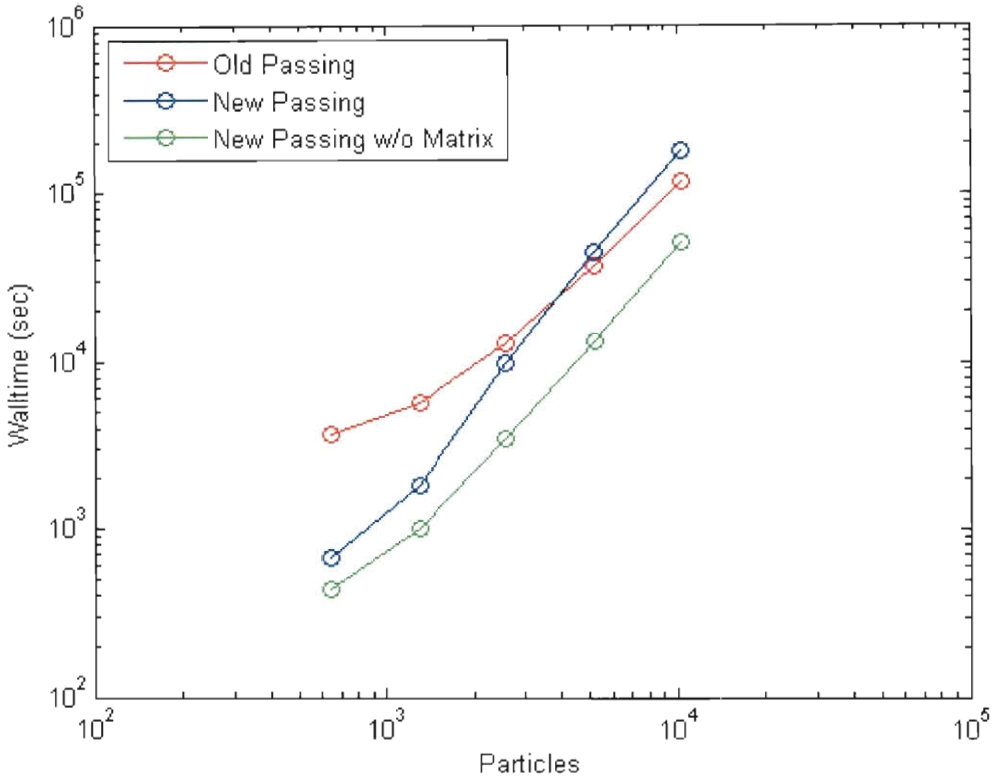


Figure 10: Runtime Comparison between Passing Routines on 20 Processors, with Matrix Formation Bypassed (P2.1)

In addition to runtime, the behavior of the simulation as the number of particles is increased is also of interest. For each Re = 1,000 run, the time required to complete ten Monte Carlo steps was recorded along with the order parameter which was noted at the end of each interval. This data allows for analysis of the convergence behavior of the system as well as looking for patterns in the time required per step.

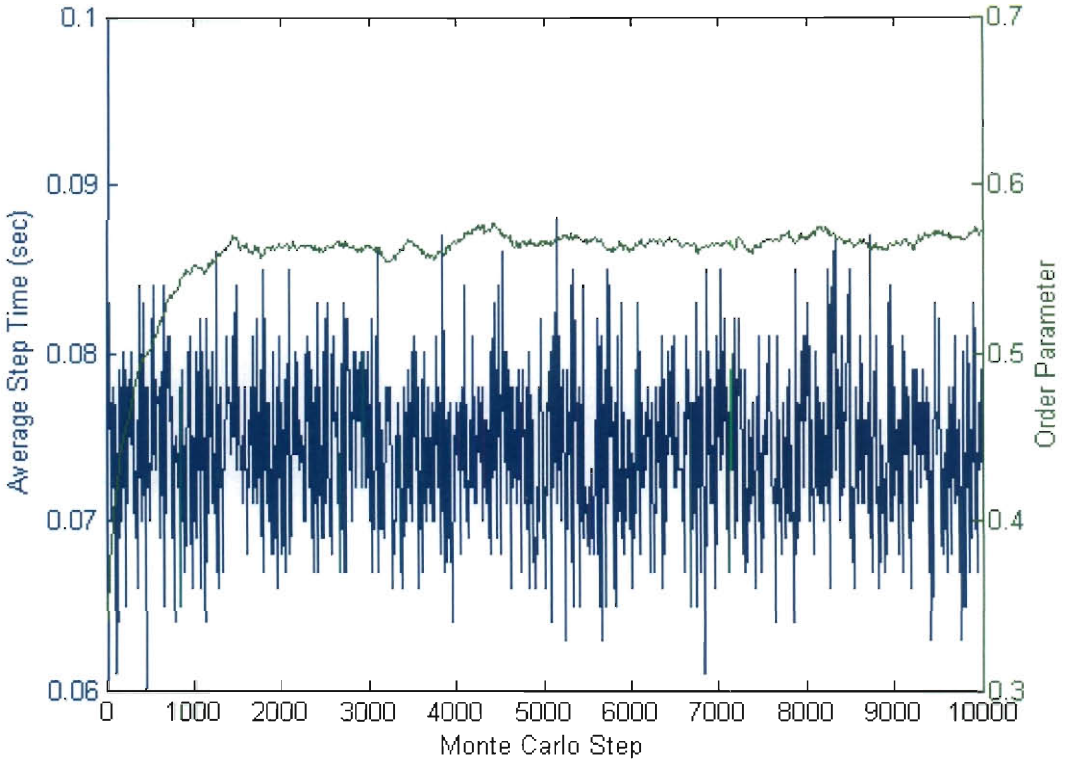


Figure 11: Convergence and Step Time for 640 Particles, *Re*=1,000

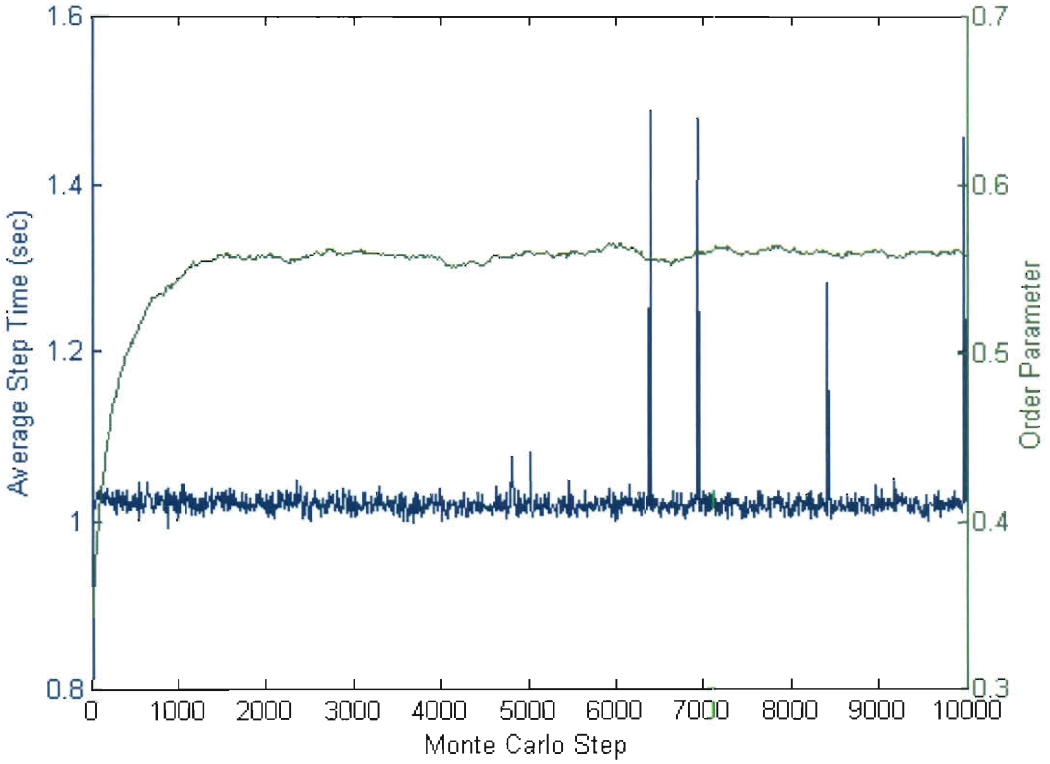


Figure 12: Convergence and Step Time for 2,560 Particles, *Re*=1,000

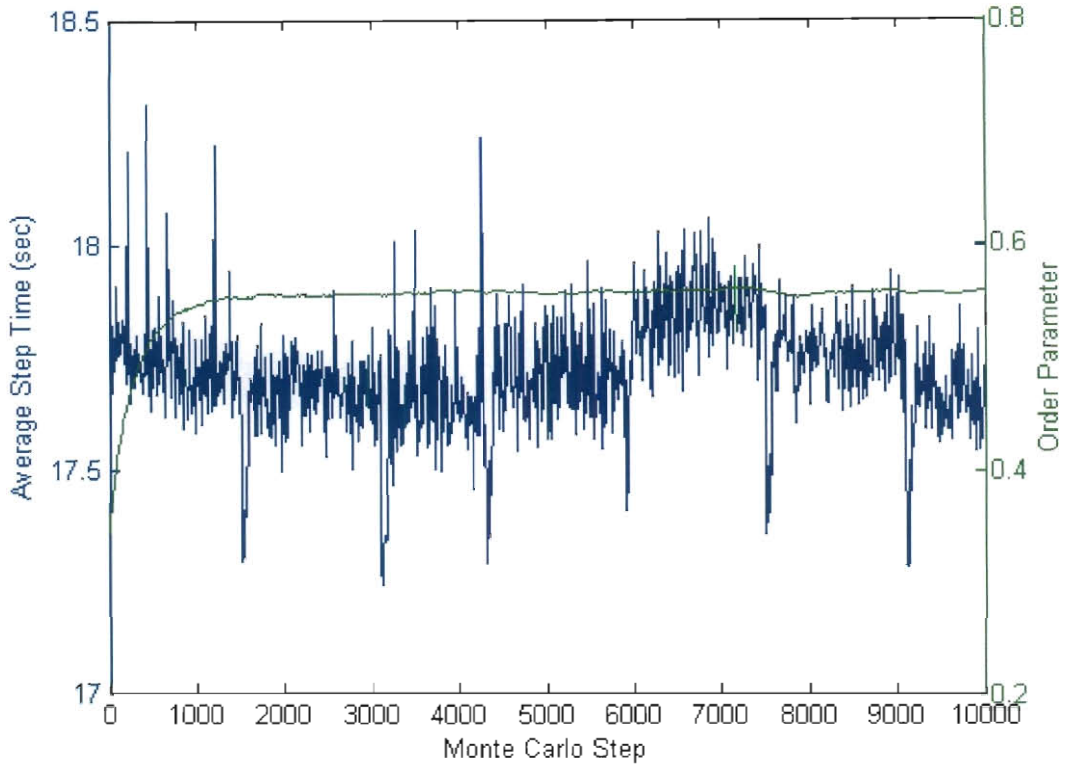


Figure 13: Convergence and Step Time for 10,240 Particles, *Re*=1,000

A few interesting observations can be made examining these plots. Regardless of the number of particles in the simulation, the systems appear to reach steady state after roughly 1200 Monte Carlo steps. However, the order parameters at the stable configuration at steady state do vary with changes in the number of particles. For smaller simulations with fewer particles, individual particles have a larger effect on the order parameter as it is based on averages and sums of the particle positions as given by equation (32). Additionally the flow field is not as easily perturbed by the motion of a single particle in larger systems with more particles, making the steady state in the 10,240 particle case much more stable than in the 640 particle case.

With respect to the time per step, there is one pattern that is readily explained. In the 10,240 system there are a series of drops in the amount of time required per step

before returning to vary about a more typical range. These drops occur during the restarts of the runs. When the system restarts the maximum step size is reset to the larger default value for the smart Monte Carlo method. As the system is already near the steady state with most particles near the wall, these large moves have a higher probability of being rejected outright. These outright rejected moves avoid the move probability acceptance calculations, and therefore the particle return cascade. These two portions of the simulation consume a substantial portion of the step time, thus skipping them speeds up the steps until the maximum step size decreases back to a value similar to that before the restart.

Chapter 4: Numerical Refinement

This chapter discusses several improvements in methods and corrections to parameters used in this study as compared to our previous work [12]. First, an improved method for calculating particle concentrations is discussed. A few errors in previous work are then discussed and improved parameter values are given. Finally the switch to a fully double precision simulation is addressed.

4.1 Concentration Profile

In our previous research the concentration profile was not optimally defined. Previously, particles were lumped into bins that were sized according to the location of the Gauss-Lobatto collocation points. Boundaries on these bins were defined in a successive fashion, with the bins closest to the walls starting the process. The distance between the wall and the closest collocation point was cut in half to define the first bin. The distance from the next collocation point to the bin boundary was then mirrored, so that the bin was the same size from the collocation point both toward the center of the channel and toward the wall. This entire process is then mirrored across the centerline of the channel. Below is an illustration of the bins that are created about the collocation points.

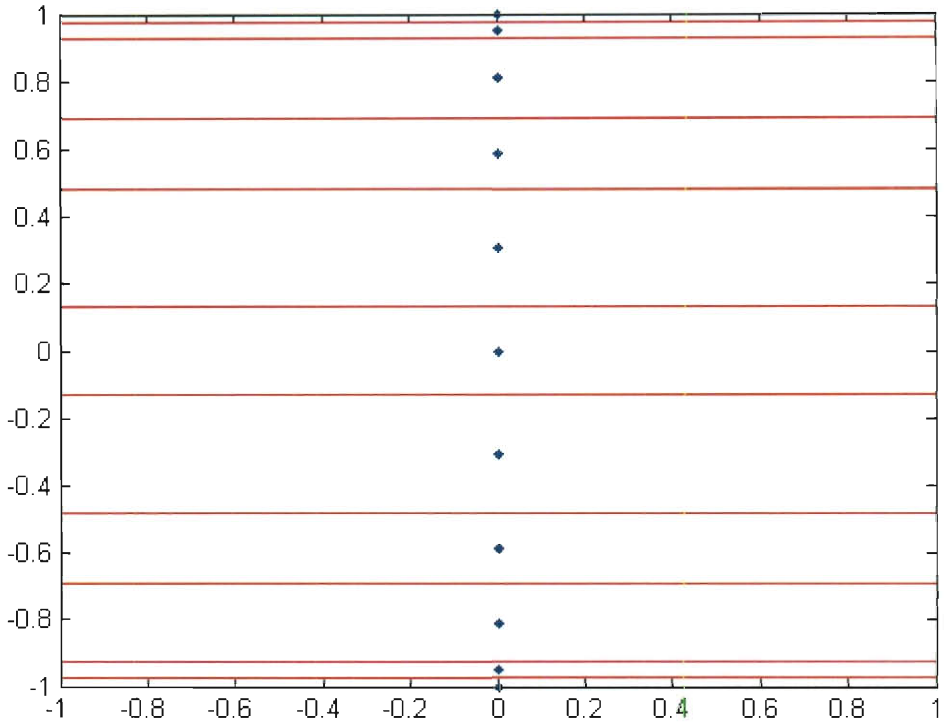


Figure 14: Previous Bins for Concentration Profile Generation

The blue dots in the above plot represent the collocation points, while the red lines indicate the boundaries of the previous concentration bins. This setup allows for the easy assignment of concentration values to each collocation point for use in the velocity profile generation. In the previous implementation, if a particle center fell within these boundaries the entire volume of the particle was considered to be within the boundaries. However, it was noted that in some cases the diameter of a particle would not fit within the height of a given bin (generally those closest to the wall). The concentration profile is now calculated in a more realistic fashion, where the actual volume of a particle present in each bin is counted in that bin. This means that a particle can be sliced and the volume of these slices placed into separate bins.

This new ability to divide particles into slices in different regions presented an issue for the previous bins system. Collocation points at the wall should always have a concentration of zero assigned to them, as the membrane surface cannot have a particle concentration. In the old system this was accounted for in the fact that the bin associated with the membrane collocation points was thinner than a single particle radius, making it unable to contain any particles under the previous concentration calculation method. Refinement of the concentration near the surface was desired, as this is the region of most interest. The previous concentration calculation method was thought to provide this refinement, however it can be seen from Figure 14 that this was not the case. Furthermore, the bins alternated between thicker and thinner bins in the previous implementation, exacerbating the influence of some nodes, and reducing the influence of others.

Here the method of determining the size of the bins was altered to provide this refinement by changing the size of the bins at the wall. These bins on the surface now have zero thickness, and are assigned a zero concentration. The other bins are created as before, with mirroring of the distance from the collocation point to the previous bin boundary. This results in the bin setup below.

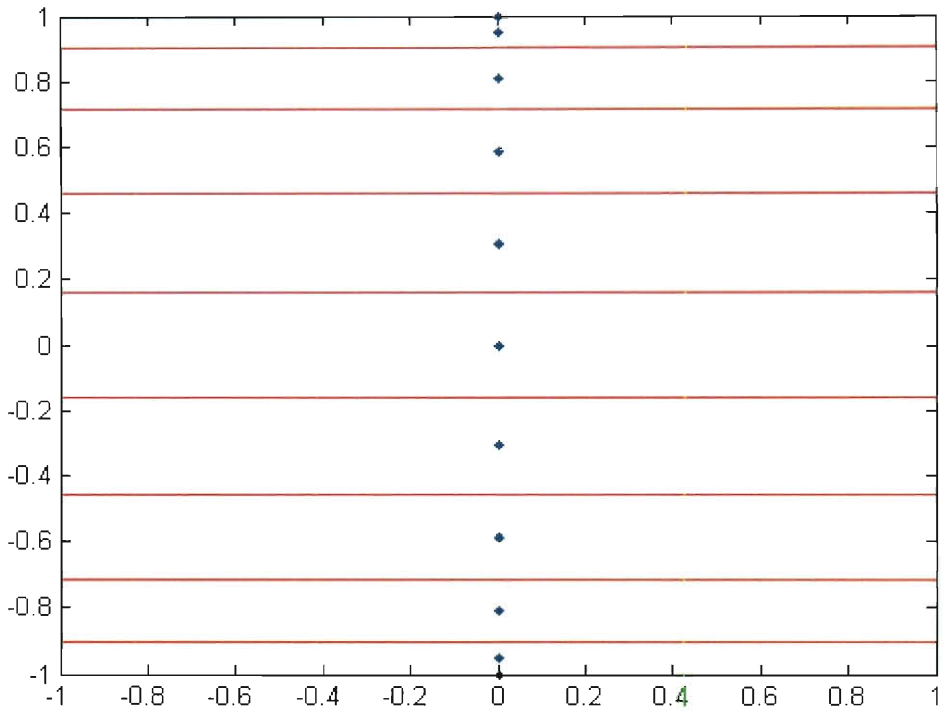


Figure 15: New Bins for Concentration Profile Generation

By examining this plot, it becomes clear this is a better way of refining the bin size so that the bins close to the membrane surface become thinner. It should be noted that the above plots are for illustrative purposes only, and contain only 11 collocation points, while the real system utilizes 25. The plots below show how the changes affect the final state of the simulation.

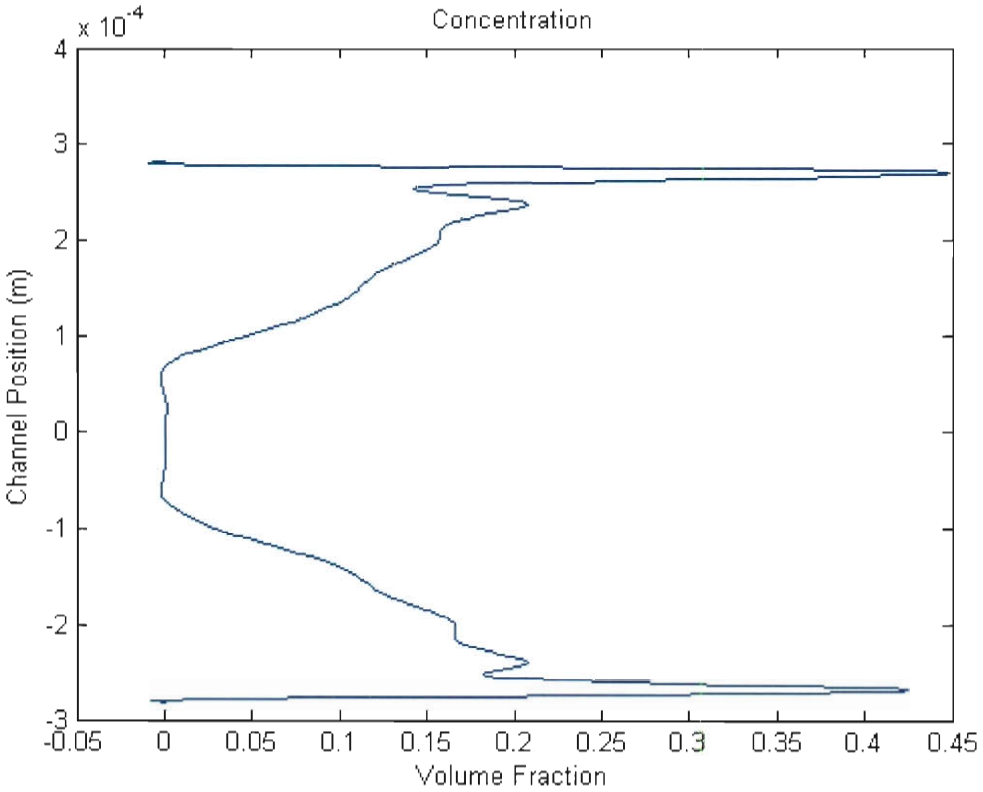


Figure 16: Old Bin Concentration Calculation, *Re*=1,000

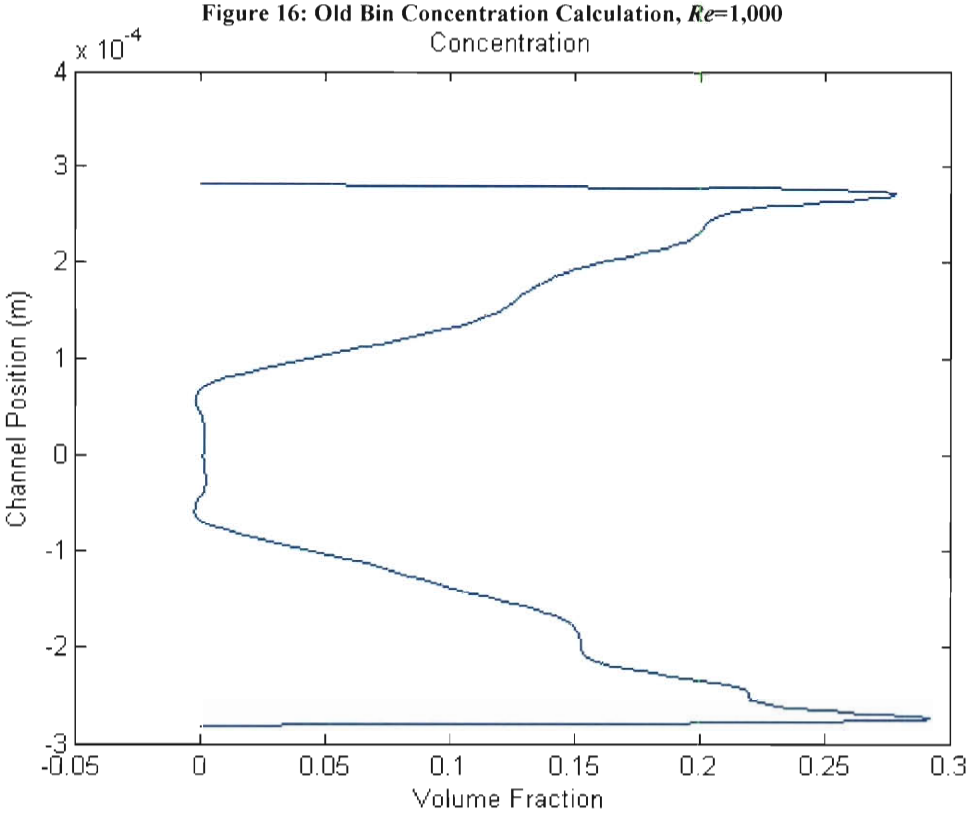


Figure 17: New Bin Concentration Calculation, *Re*=1,000

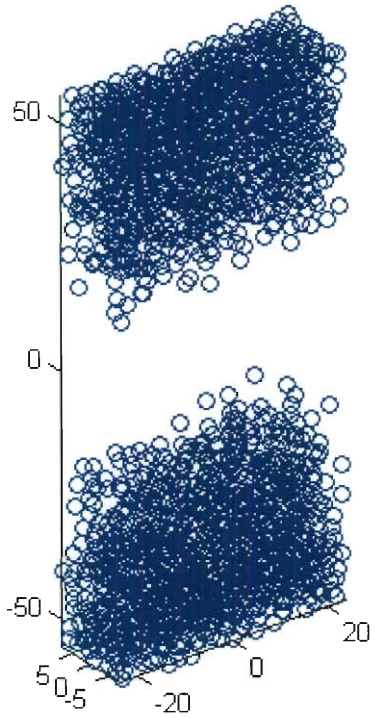


Figure 18: Final State of Simulation, *Re*=1,000

Figure 17 appears to fit better with the final state of the simulation, illustrated in Figure 18. It doesn't contain extreme reversals in the concentration profile, the concentration remains positive at all points in the flow, and it has an element of smoothness that was not previously present. Perhaps the most interesting aspect of this change is that it has a limited effect on the outcome of the simulation in terms of the final maximum velocity and order parameter. In the old bin calculation scheme, the maximum velocity was roughly 1.49 m/s with an order parameter of 0.590, while the new concentration calculation results in a maximum velocity of 1.47 m/s, and an order parameter of 0.575. These are approximate values, as the final state oscillates about an estimated mean value. Finally, one added bonus, although small, is that the new concentration calculations actually take less time. The runtime for the new setup takes

158 minutes, while the old calculations take 160 minutes. This is small savings, but it is consistently present across multiple runs.

4.2 Corrections to Previous Errors

Several small errors were noticed during this round of improvements to the simulation. The most notable was an errant calculation of the distance from the surface of the particle to the surface of the membrane in the particle-surface electrostatic double layer energy calculation. This led to the strength of this interaction being over stated, which forced particles back toward the core of the flow. This error has been corrected in the preceding two chapters and for all future work. Another error was in the calculation of the distance between particles for the electrostatic double layer potential. In this case, the distance from the surface of the particle to the shear plane was neglected, this distance is important when using zeta potentials rather than surface potentials. This error, however, had a negligibly small effect.

Additional errors include the use of improper values for the concentration of electrolytes in the flow, as well as several other electrical parameters. The table below compares the previous values [12] used to the new values.

Simulation Parameters	Old Parameters	New Parameters
Particle Volume Fraction	0.1	0.1
Number of Particles	2,100	2,100
Particle Radius (BSA), R	3.13 nm	5.0 μm
Permeate Flux Velocity	30 μm/s	30 μm/s
Cell Size Ratio (WxHxL)	1x7x3	1x7x3
Temperature, T	300 K	300 K
Pure Water Density	996.63 & 1000 kg/m^3	996.63 kg/m^3
Pure Water Viscosity, μ	0.8887 & 0.85659 cP	0.8887 cP
Maximum Steps	10,000	10,000
Particle Valence, z_p	-20	----------
Electrolyte Concentration, C_{EL}	10^{-7} M	10^{-3} M
Particle-Particle Hamaker Constant, $A_{H\text{-}SS}$	1.65 X 10^{-21} J	1.65 X 10^{-21} J
Particle Zeta Potential, ζ	-30 mV	-30 mV
Solvent Dielectric Constant, ε_r	78.54	78.54
Particle-Wall Hamaker Constant, $A_{H\text{-}SP}$	1.65 X 10^{-21} J	1.65 X 10^{-21} J
Wall Valence, z_p	-20	----------
Wall Zeta Potential, ζ	-30 mV	-30 mV
Electrolyte Valence, z_e	-20	1

Table 1: Comparison of Old and New Simulation Parameters

The original valence values were guesses and particle valence was incorrectly used in equations (7), (10)-(12), and (18) in place of the electrolyte valence. Particle valanece has been eliminated from these equations and changed to the electrolyte valence. The new value for the electrolyte valence fits with real world values seen in literature for water [27]. One interesting result of these changes is that they affect the Debye screening length, which in turn changes the validity of the equations used for the electrostatic double layer energy calculation. The old values resulted in a κa_i of roughly 0.065, while the new parameters result in a value of approximately 0.32. This moves the work closer to the validity range of both of the equations that are used for the electrostatic double layer energy, although formal validity remains an order of magnitude off (see equations (13) and (14)). It should be noted that for the above work in Chapter 3 the

particle radius is generally taken to be 5 μm, which is a very large particle compared to the radius of some standard foulants, such as BSA, which has a radius of 3.13 nm. This large particle radius puts this work well within the validity region with a κa_i value of 517. For this reason, the fictional particle radius of 5 μm is assumed for the remainder of the work.

An additional fix between the old simulation and the new simulation addresses two different densities and viscosities of pure water used in different aspects of the simulation. The first of these values were used in the legacy code in the original force bias equations, and to generate the pressure differential that drives the flow. However, these densities and viscosities were slightly different from those used to generate the flow field. These values are now consistent in the latest version of the simulation.

4.3 Double Precision

Single precision is all that is required for a Monte Carlo simulation, as has been documented [39]. Previously, the simulation was a mix of single and double precision calculations, which while acceptable, could result in portability issues. The current simulation utilizes double precision calculations exclusively to remove any ambiguity, with very little change to the final state of the parameters of interest. For a Reynolds number of 1,000 with mixed precision calculations, a final state of approximately 0.570 for the order parameter and 1.55 m/s for the maximum velocity in the channel is obtained. In the purely double precision simulation these values are approximately 0.569 and 1.55 m/s. As before, it should be noted that oscillations in the data make it difficult to obtain an exact value for either of these quantities, and these presented values represent an approximate median for the final state of the output. Given all of these

changes, the following velocity profile development was obtained for a Reynolds number of 1,000. For comparison purposes, the parabolic Newtonian velocity profile with no particle concentration effects on viscosity (*i.e.* the viscosity of clean water is taken throughout the domain) is also illustrated in this plot.

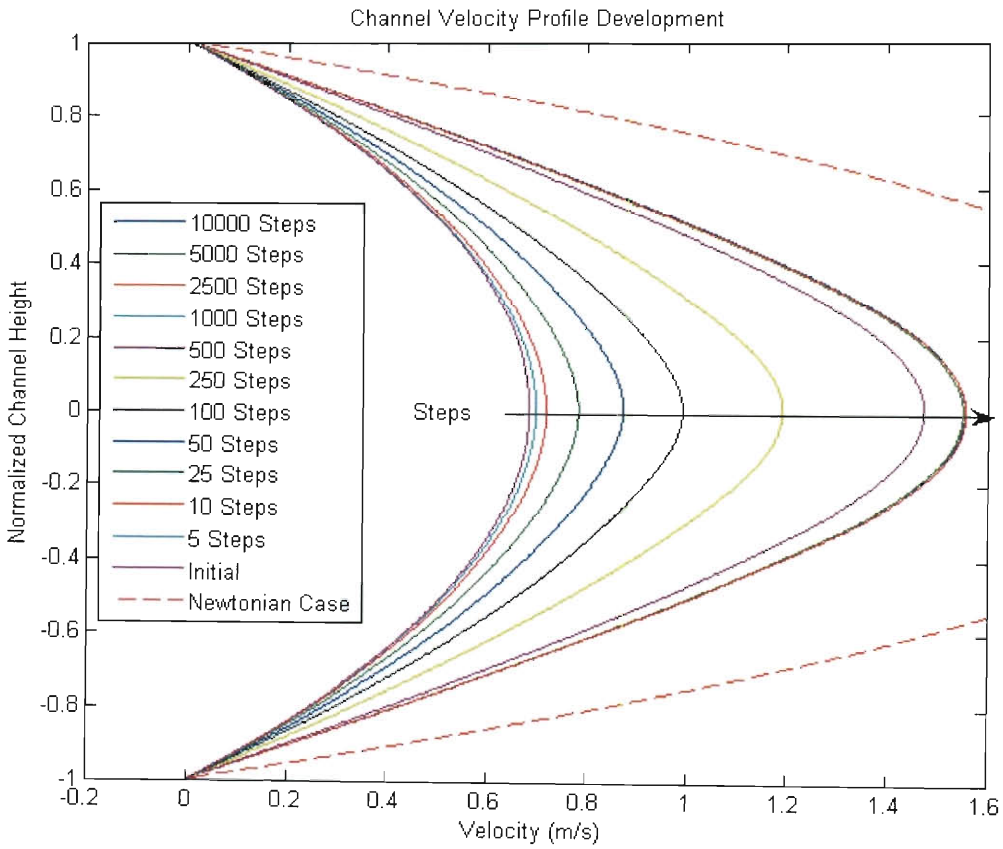


Figure 19: New Velocity Profile Development, *Re*=1,000

Chapter 5: Surface Roughness

Previously, the surface of the membrane was assumed to be perfectly smooth. The membrane surface had no variation and the particles in the flow saw the same surface regardless of their position in the channel. In reality, membranes are imperfect, both by design and due to manufacturing. In order for the membrane to be semi-permeable it must have pores, which the clean water may pass through. The roughness also depends on the material utilized to manufacture the membrane and can contribute heavily to the interaction the particles have with the surface [40].

For these reasons, surface roughness was added to the simulation. The method through which it was introduced is straight forward; prior to initializing the particles in the channel a number of particles are fixed directly to the surface of the membrane. These particles remain fixed throughout the Monte Carlo steps and can be assigned a variety of electric properties. The membrane properties were assigned to these particles in this work, as they are meant to model roughness of the membrane surface.

One critical question was whether or not the particles on the surface should be arranged in an ordered fashion, like a grid, or if they should be placed randomly. Ultimately, it was decided that, based on the atomic-force microscopy images of membrane surfaces in Elimelech and Zhu's paper [40] that the particles should be placed randomly. While the image of the composite membrane appears to exhibit an ordered roughness, it should be noted that not all peaks are the same height. The cellulose membrane image depicts a strong variation in the size of the peaks, and demonstrates very little order.

The number of particles fixed to the surface was varied from 1 particle each on the top and bottom, to 100 particles each on the top and bottom surfaces. The particles on the top surface were placed independently of the particles placed on the bottom surface. For a volume fraction of 10% with 2,100 particles and a channel with size ratio of 1W x 3L x 7H the area exposed to the flow on one surface is approximately 780 square units, where a unit is defined as a particle radius. Thus for highly ordered surface roughness, where each particle is given a 2x2 box to rest in, the maximum number of particles that could fit on one surface is 195.

Surface roughness can be measured using a variety of parameters, however the most popular is the arithmetic mean roughness, given by the parameter [41]:

$$R_a = \frac{1}{l_L l_W} \int_0^{l_L} \int_0^{l_W} u(W,L)\, dW\, dL \tag{38}$$

Where $u(W,L)$ is the surface function, l_W and l_L are the dimensions of the surface (scaled so that the radius of a particle is 1), and R_a is the roughness parameter. For each particle that is fixed to the surface there is a roughness of $5\pi/3$ added to the surface. This roughness comes from the top portion of the sphere plus the cylinder below the hemisphere, as seen when looking down on the membrane surface. The mean surface height must then be calculated, based on the number of particles fixed to the surface. Finally, a cylinder of radius 1 with height set to the mean surface height is subtracted from $5\pi/3$ to determine the surface roughness contribution from the particles (assuming the mean surface height is less than or equal to 1). The remainder of the surface area is at the negative of the mean surface height. This procedure is outlined in the two equations below.

$$\mu_m = \frac{N_s 5\pi/3}{l_W l_L} \tag{39}$$

$$R_a = \frac{(l_W l_L - N_s \pi)\mu_m + (5\pi/3 - \mu_m \pi) N_s}{l_W l_L} \tag{40}$$

where N_s is number of surface particles, and μ_m is the mean surface height. Utilizing these relationships, the surface roughness parameter for each test case run is shown in the table below.

Surface Particles	Mean Surface Height	Surface Roughness R_a
1	0.006717	0.01338
5	0.03358	0.06581
25	0.1679	0.3020
50	0.3358	0.5363
100	0.6717	0.8020

Table 2: Surface Roughness Parameters

These different surface roughness cases were run for Reynolds numbers of 0.1, 10, 100, and 1,000. The results of these simulations are illustrated below.

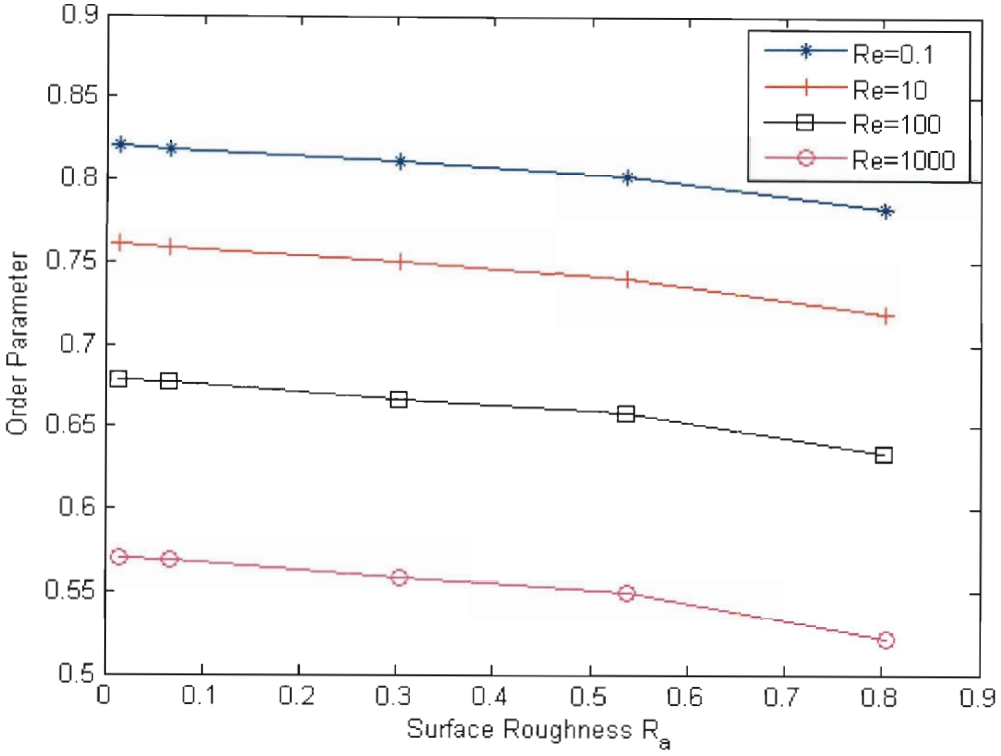


Figure 20: Surface Roughness Vs. Order Parameter, 10% Volume Fraction, 5 µm Particle Radius

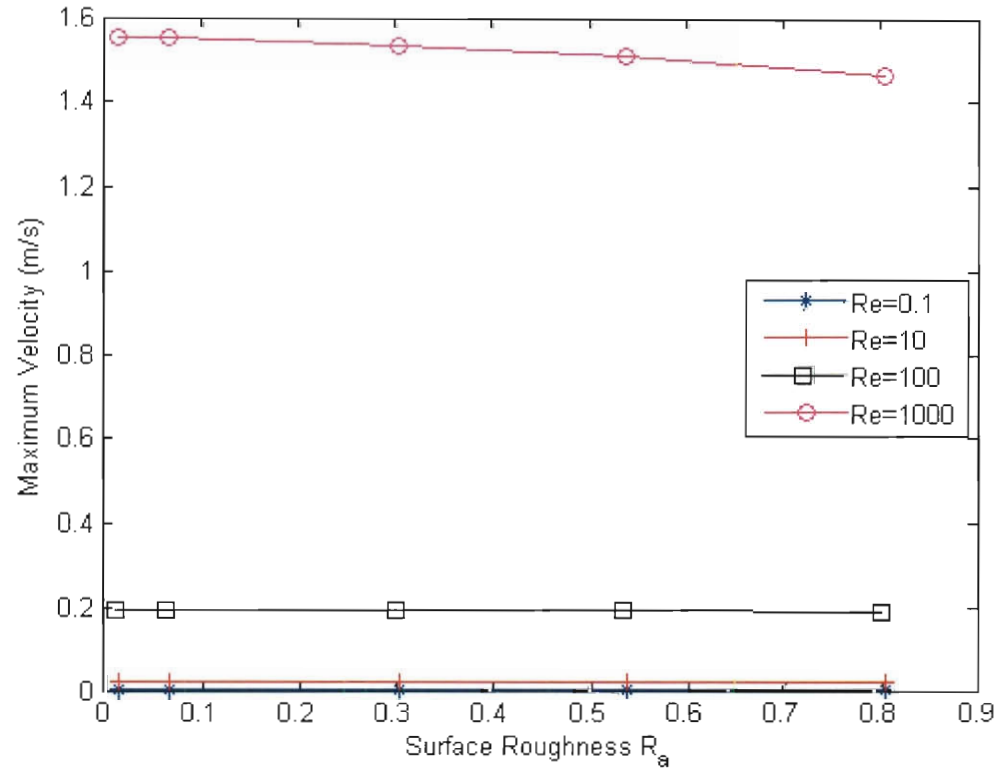


Figure 21: Surface Roughness Vs. Crossflow Velocity, 10% Volume Fraction, 5 µm Particle Radius

As the surface roughness increases from 0.01338 to 0.8020 the order parameter at Reynolds number of 1,000 drops from approximately 0.57 to roughly 0.52. Similarly, for Reynolds number of 0.1 the order parameter drops from 0.8193 to 0.7811 following a comparable profile. The relationship between maximum velocity and surface roughness at Reynolds number of 0.1 isn't very clear, but it appears to decrease at a slightly more than linear rate for Reynolds number of 1,000. The surface roughness significantly affects the final state of the simulation, and is an important variable to consider when attempting to obtain an accurate simulation.

The above particles fixed to the wall were assumed to have the same charge as the membrane and particles in the flow. Should the particles be oppositely charged, it would be expected that the particles in the flow would be drawn to the surface. The comparison of the order parameter for the oppositely charged case and the equally charged case at a Reynolds number of 1,000 is illustrated below.

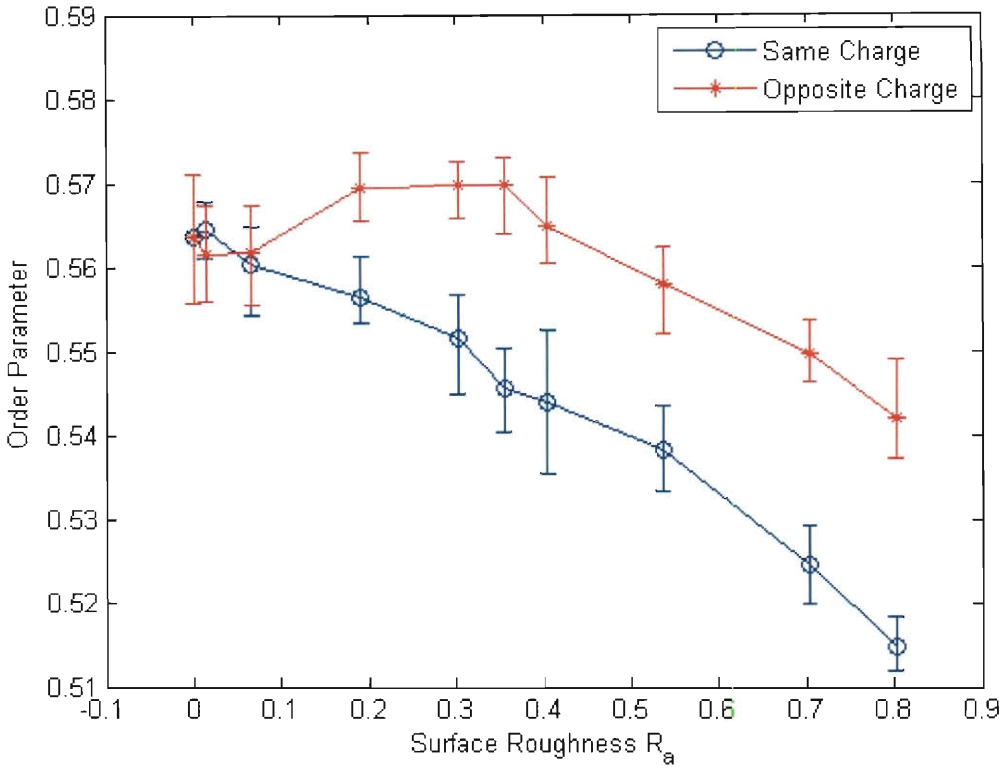


Figure 22: Comparison of Order Parameter for Equally and Oppositely Charged Surface Particles, 10% Volume Fraction, 5 µm Particle Radius

The error bars indicates the minimum and maximum order parameter in the last 2,000 Monte Carlo steps, and the point represents the mean order parameter during those steps. An interesting feature of this plot is that for the oppositely charged case the initial increased draw to the membrane surface can be seen with increasing surface particle count. However, for surface roughness above 0.35, the order parameter begins to drop off, but always remains higher then in the equally charged case. Thus, for the oppositely charged case, more particles are drawn to the wall than the equally charged case, for all surface roughnesses. The drop-off observed in both higher roughness cases can be explained because the surface eventually becomes saturated with fixed particles, which are not considered in the order parameter calculation, as illustrated below.

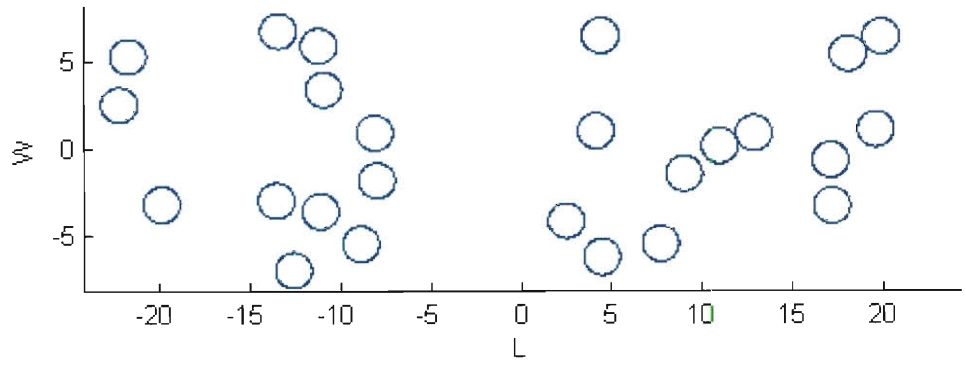


Figure 23: A Sample Case of 25 Particles ($R_a = 0.3020$) Attached to the Membrane Surface

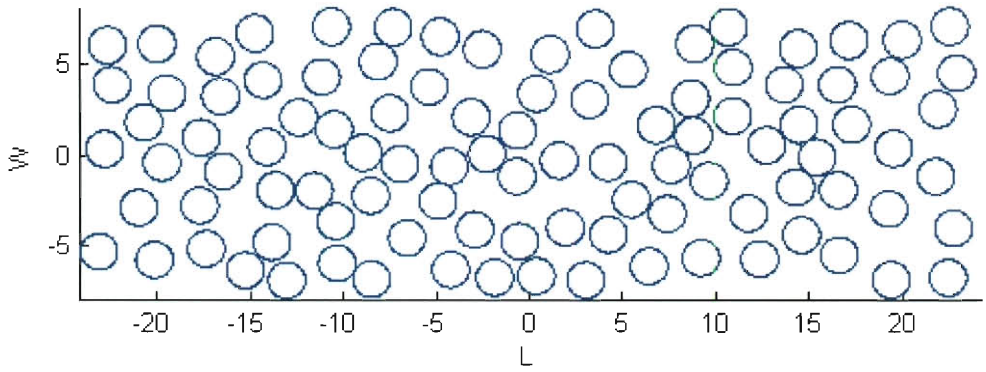


Figure 24: A Sample Case of 100 Particles ($R_a = 0.8020$) Attached to the Membrane Surface

The decreasing free space near the surface makes it difficult for the flow particles to get near the surface. For the maximum roughness case there is very little available space for the flow particles to approach the wall, which results in the concentration profiles for both the equally and oppositely charged particles of these cases shown below. The core of the flow appears to be similar in both cases; however, there is a noticible difference near the walls. In these cases the oppositely charged particles attract many more particles from the flow toward the membrane surface, while the similarly charged particles repel particles back into the flow.

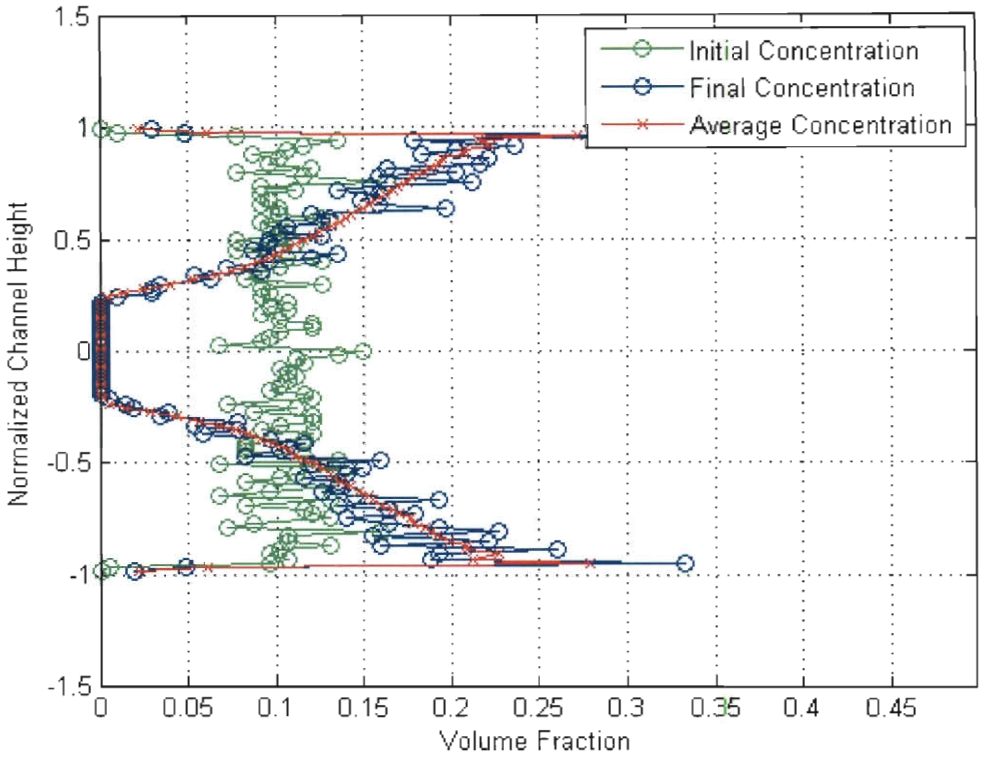


Figure 25: Equally Charged Particles, Maximum Roughness, *Re*=1,000, 10% Volume Fraction, 5 µm Particle Radius

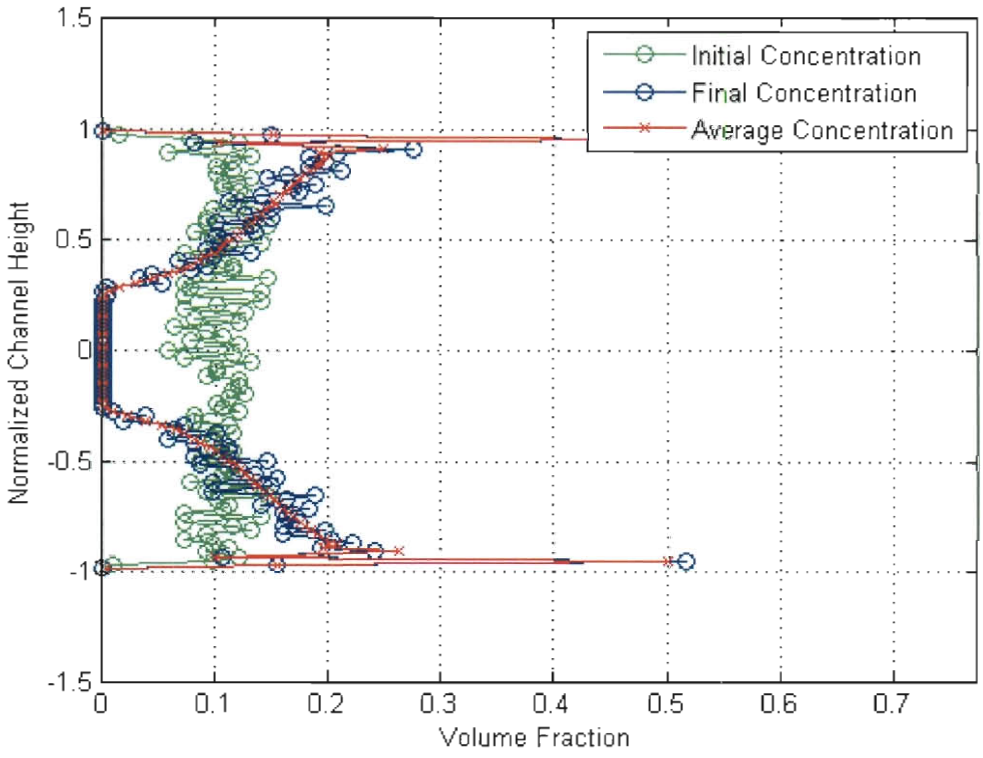


Figure 26: Oppositely Charged Surface Particles, Maximum Roughness, *Re*=1,000, 10% Volume Fraction, 5 µm Particle Radius

Chapter 6: Alternate Electric Interactions

There are many ways to calculate the electrostatic double layer interaction between two particles. These formulas depend on the separation of the particles and their electric properties. When examining the two-range solution used in previous chapters, both a close-range electrostatic double layer potential formula (equation (10)) and a long-range formula (equation (12)) are used. With the "New Parameters" from Table 1 above, the close range interaction can only be used when the particles are separated by a distance much less than 5 μm, and the long range interaction formula can be used when the particles are separated by at least -9.9903 μm, where the negative value indicates validity for all ranges. However, for a particle of the size of BSA, with radius of 3.13 nm, the close range formula can only be used for a separation much less than 3.13 nm, and the long range can only be used for a separation much greater than 3.41 nm, leaving a gap in the formula coverage. This neglects the additional requirements on the product of the inverse Debye screening length and the particle radius.

To test whether or not these formulae produce reasonable outputs even though they contain this gap in coverage, an additional electrostatic double layer potential formula was added to the simulation. This formula is intended for use at all separation ranges, the cost being that it is a substantially more complicated and computationally expensive calculation. In general form, this formula is given as [20]:

$$V_{EDL-SS} = \frac{\pi\varepsilon_0\varepsilon_r k_b^{\,2}T^2}{z_p^{\,2}e^2}\frac{a_1a_2(z-a_2)(z-a_1)}{z\left[(a_1+a_2)z-a_1^{\,2}-a_2^{\,2}\right]}\left[\Omega_1\ln(1+\Gamma)+\Omega_2\ln(1-\Gamma)\right] \tag{41}$$

Where the Ω and Γ terms are additional functions of the size and electric properties of the particles.

$$\Omega_1 = \Phi_1^2 + \Phi_2^2 + \Lambda\Phi_1\Phi_2$$
$$\Omega_1 = \Phi_1^2 + \Phi_2^2 - \Lambda\Phi_1\Phi_2$$
$$\Phi_i = \frac{z_e e\zeta_i}{k_b T} \tag{42}$$
$$\Lambda = \sqrt{\frac{a_1(z-a_1)}{a_2(z-a_2)}} + \sqrt{\frac{a_2(z-a_2)}{a_1(z-a_1)}}$$
$$\Gamma = \sqrt{\frac{a_1 a_2}{(z-a_1)(z-a_2)}} e^{-\kappa(a_1+a_2+z)}$$

Equation (41) simplifies for identical particles to:

$$V_{EDL-SS} = \frac{2\pi\varepsilon_0\varepsilon_r R(s-1)\zeta^2}{s} \ln\left[1 + \frac{1}{s-1}\sqrt{e^{\kappa R(2-s)}}\right] \tag{43}$$

where s is the scaled center to center distance between the particles. It should be noted that this calculation may be slightly off, as ideally Φ_i would be calculated using the surface potential of the particle, however this work utilizes the zeta potential. As noted is Chapter 4.2, neglecting this difference generates very little error, but should be mentioned. When these two approaches are used for the particles of 5 μm radius the results are very similar. This should be expected, as there was no gap in the validity region for the two range solution for particles of this size. For the standard simulation at Reynolds number of 1,000, the two-range solution provides a maximum velocity of 1.55 m/s and an order parameter of 0.57. The all-range solution provides a maximum velocity of 1.53 m/s and an order parameter of 0.56. The all-range solution is only slightly more costly in its runtime, consuming an additional 255 seconds versus the two-range solution. This difference is small as the two-rnage case requires a time consuming if statement to be evaluated to determine which formula to use, while the all-range solution is more computationally expensive but does not require an if statement.

When the simulation is run with smaller particles of radius 3.13 nm, there is a more noticeable difference. Below are the two graphs of the concentration in the channel that result from the use of these two methods.

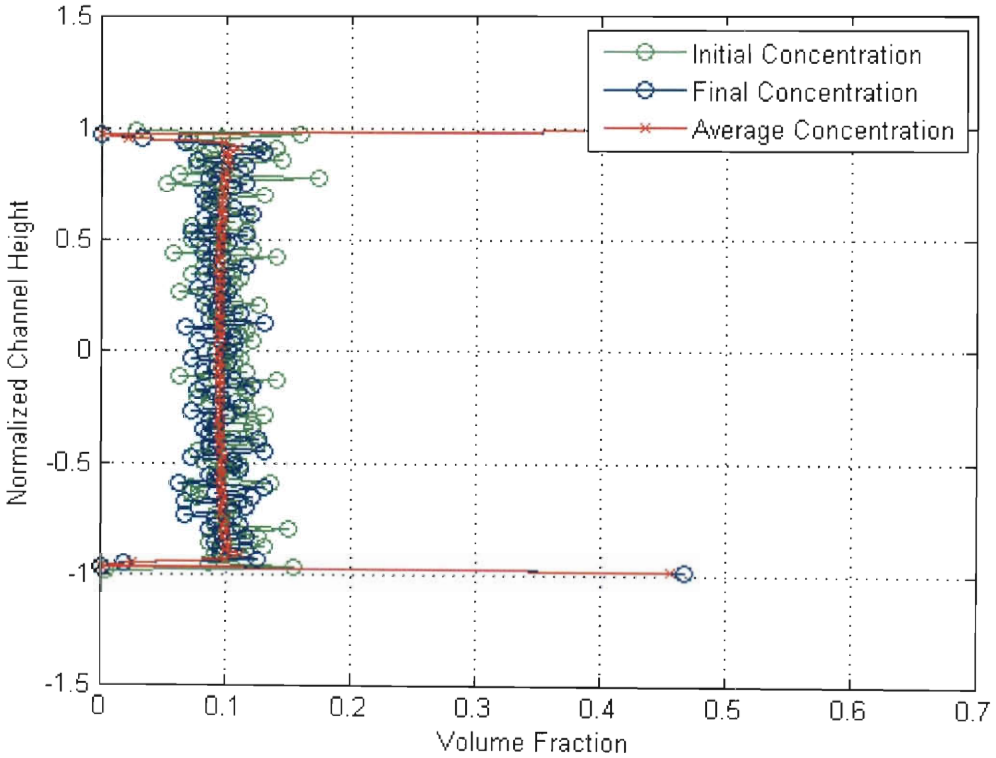


Figure 27: Two-Range Solution, R=3.13 nm, *Re*=1.0

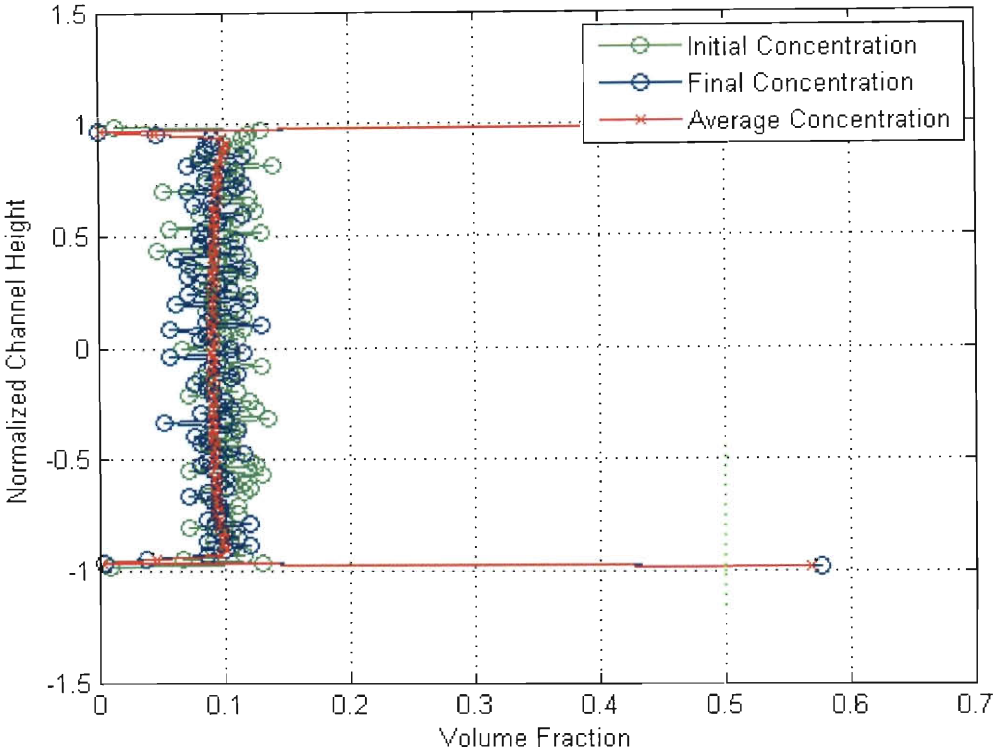


Figure 28: All-Range Solution, R=3.13 nm, *Re*=1.0

In both cases there is a clear double layer effect, with a high concentration directly on the wall and a near zero concentration just displaced form it. The difference that is worth noting is that in the all range case, the concentration returns to the core flow concentration gradually as the distance from the walls increase. In the two-range case, there is a slightly lower concentration directly adjacent to the wall, and two bins are still required to return to the core concentration. However, the concentration gradient in the two-range case is much greater from the second bin to the core than in the all-range case. With the two-range solution, run with a Reynolds number of 1.0, the maximum velocity in the channel is 1.18 m/s with an order parameter of roughly 0.37. Utilizing the all-range solution, the maximum velocity in the channel is 1.22 m/s, and has an order parameter of 0.38. The two-range solution appears to remain a valid approximation, and

does save a little time, dropping the runtime by 100 seconds. However, should additional accuracy be desired, the all-range solution is probably more appropriate as it does not require significantly more runtime.

Chapter 7: Future Work

7.1 Fast Multipole Method

The new P2 and P2.1 implementation opens the possibility for additional accelerations in the simulation. Since the systems is treated on a system state basis rather than an individual particle state basis, methods for calculating the potential field in the entire channel become viable. This may sound far more arduous than calculating the individual particles electric potentials, as the entire potential field must be formulated, however, many methods exist that make the later significantly easier than the former for large particle systems.

One of the methods available is the Fast Multipole Method (FMM). This method was developed by Leslie Greengard specifically for the purpose of calculating potential fields [19]. This method divides the channel into smaller blocks, based on the density of particles in the region. If there are many particles tightly packed in small regions the blocks become small. In low density regions they become large. Distant particles see these blocks as lumped particles with the net electric characteristics of the group of particles, rather than individual particles. In the near-field, particles still see each other as before, but in the far-field the local particle sees blocks as lumped particles with different electric characteristics than individual particles. Clearly, this sounds like a powerful method, as the number of calculation is reduced from the $O\left(n^2\right)$ scaling in the current implementation. However, there is one substantial drawback to this method.

The breakeven point for applying this method versus the current particle energy calculation method occurs only when there are more than 150,000 particles in the system

[42]. Currently, the system is running at a maximum of 10,240 particles, and in the best parallel implementation thus far, this takes approximately 14 hours of runtime to reach 10,000 MC steps. It should be clear then that 150,000 particles cannot be reasonably achieved in this current implementation. Should the parallel runtime for this problem improve, or the maximum time allowed per simulation increase, FMMshould perhaps be considered. For the time being, FMM was not employed in this simulation.

7.2 Additional Refinements

The current simulation has several points at which memory management could likely be improved. Memory for the system is allocated at the very beginning of the work and never deallocated as the simulation progresses. To increase the size of the simulation in the future, dynamic memory allocation and deallocation will be necessary.

Advanced analysis of time consumption of the individual subroutines in this work through profiling could also identify points at which the simulation reaches a bottleneck. This would allow for focused diagnosis of the most troublesome points in the simulation and provide guidance on where improvements (better adapting a routine to the architecture or finding innovative approaches to calculations in the subroutines) would generate the most runtime savings.

Finally, in addition to speed, there will always be room to improve upon the physics in the simulation. Continued refinement of the electric interactions will lead to more accurate results. Special case formulas for the very small particle regime and other real world conditions can provide useful information for experimentalists. Two cases are of particular interest: (1) does fouling occur for very low flow rates (both cross and

permeate flow) and (2) what is the effect of different size surface roughness, particularly surface roughness that is much larger than the size of the particulate in the flow.

Chapter 8: Conclusions

This work has improved upon the speed of the previous simulation (P1) for particle counts of 2,100 while maintaining accurate results. This has been accomplished by adopting a new method of handling the particle data and redesigning the mode in which the simulation utilizes parallel processing in implementation P2 and P2.1. In addition to improved runtime, additional tools and fixes have been added. The concentration profile is more realistic and depends on the actual volume of a particle in a given bin, rather than a lumped particle volume. The ability to fix particles to the membrane surface has been added to address the need to simulate surface roughness. More appropriate electric values have been added for the particles, as has a new all-range electrostatic double layer potential formula.

Several concerns raised by the previous work have been addressed. The previous method in which the particles moved has been altered to force all particles to attempt to move at once, rather than in sequence. This has changed the output slightly, but by a very small percentage. This change has also opened the door to utilizing a new particle potential evaluation scheme that while not well suited for the current work, could be valuable in the future for simulations of hundreds of thousands of particles.

There had been some questions about the electrostatic double layer interaction being overly strong. Parameters used in the simulation were refined, and an additional method for calculating the electrostatic double layer potential was added. The work has shown that even with corrected electric values, the electrostatic double layer between the particles fixed to the wall and the particles in the core of the flow is still discernable. The additional all-range electrostatic double layer formula allows a check of the simulation,

and confirms that this is not an artifact of breaking the validity of the two-range equations.

Surface roughness has been shown to significantly influence the flow. High surface roughness can force particles back into the core flow, and results in a reduced flow rate. When comparing the flow effect with the surface roughness parameter R_a, a slightly greater than linear effect can be seen on the flow velocity and the order parameter. While the effect of surface roughness on the order parameter appears to be independent of the crossflow Reynolds number, the effect of surface roughness on crossflow velocity appears to most strongly affect high Reynolds number flows.

Upon visualizing the motion of the flow, a variety of conculsions can be readily made. In low Reynolds number crossflow, few particles are swept from the wall and mixed with the particles originally at the center of the flow.

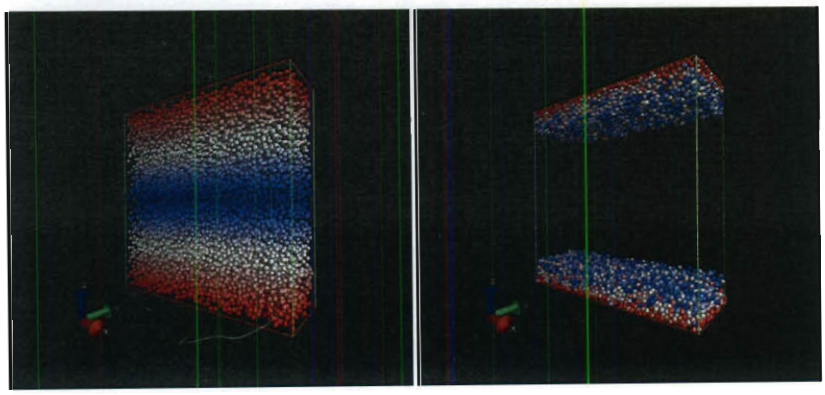


Figure 29: Initial and Final States for 10,240 Particle Simulation at *Re*=10, Volume Fraction 10%

Similarly, particles that were originally at the center of the flow tend to remain closer to the center and do not mix much with particles closer to the wall. The simulation with a low Reynolds number appears to simply compress the particles to the wall. Only

particles close to the core of the flow are likely to move with the crossflow, while particles near the wall barely move in any direction.

For high Reynolds number flows, the particles mix freely. Particles originally in the core of the flow can migrate to the wall, and particles originally at the wall are swept into the crossflow.

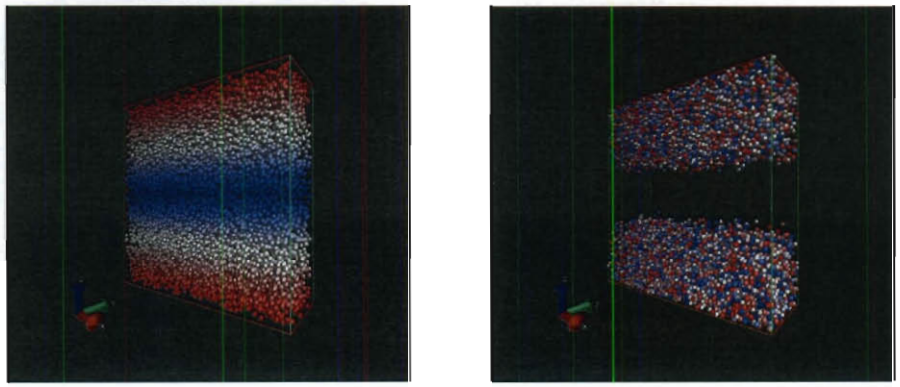


Figure 30: Initial and Final States for 10,240 Particle Simulation at *Re*=1,000, Volume Fraction 10%

Unlike the low Reynolds number cases, high Reynolds number cases do not simply compress particles to the wall as the simulation progresses. The mixing continues throughout the simulation, and while particles closer to the core of the flow move with the crossflow more readily, plarticles near the wall can still be observed moving downstream. This indicates that, as one would expect, higher Reynolds numbers tend to inhibit the formation of a foulant cake layer on the membrane surface.

References

1. *Water, sanitation and hygiene links to health.* 2004 [cited 2008; Available from: http://www.who.int/water_sanitation_health/publications/facts2004/en/index.html .
2. *Progress on Sanitation and Drinking-Water: 2010 Update*, W.H. Organization, Editor. 2010.
3. *Reverse Osmosis (RO) Water Filters.* [cited 2011 2/12/2011]; Available from: http://www.home-water-purifiers-and-filters.com/reverse-osmosis-filter.php.
4. Mallevialle, J., P.E. Odendaal, and M.R. Wiesner, *Water Treatment Membrane Processes*. 1996, New York, NY: McGraw-Hill.
5. Flemming, H.C., G. Schaule, T. Griebe, T. Schmitt, and A. Tamachkiarowa, *Biofouling-the Achilles heel of membrane processes.* Water Science and Technology, 1998. **38**(8-9): p. 291-298.
6. Visvanathan, C. and R.B. aïm, *Studies on colloidal membrane fouling mechanisms in crossflow microfiltration* Journal of Membrane Science, 1989. **45**(1-2): p. 3-15.
7. Harris, A., *A Mechanistic Study on the Coupled Organic and Colloidal Fouling of Nanofiltration Membranes*, in *Civil and Environmental Engineering*. 2008, Rice University: Houston, TX.
8. Vrouwenvelder, J.S., S.M. Bakker, M. Cauchard, R.L. Grand, M. Apacandie´, M. Idrissi, S. Lagrave, L.P. Wessels, J.A.M.v. Paassen, J.C. Kruithof, and M.C.M.v. Loosdrecht, *The membrane fouling simulator: a suitable tool for prediction and characterisation of membrane fouling.* Water Science and Technology, 2007. **55**(8-9): p. 197-205.
9. Xu, P., C. Bellona, and J.E. Drewes, *Fouling of nanofiltration and reverse osmosis membranes during municipal wastewater reclamation: Membrane autopsy results from pilot-scale investigations.* Journal of Membrane Science, 2010. **353**(1-2): p. 111-121.
10. Bai, R. and H.F. Leow, *Microfiltration of activated sludge wastewater—the effect of system operation parameters.* Separation and Purification Technology, 2002. **29**(2): p. 189-198.
11. Zaw, H.M., T. Li, and H. Nagaoka, *Simulation of membrane fouling considering mixed liquor viscosity and variation of shear stress on membrane surface.* Water Science and Technology, 2011. **63**(2): p. 270-275.
12. Boyle, P., *Coupling a Dynamically Updating Velocity Profile and Electric Field Interactions with Force Bias Monte Carlo Methods to Simulate Colloidal Fouling in Membrane Filtration*, in *Mechanical Engineering and Material Science*. 2009, Rice University: Houston, TX. p. 126.
13. Vrijenhoek, E.M., S. Hong, and M. Elimelech, *Influence of membrane surface properties on initial rate of colloidal fouling of reverse osmosis and nanofiltration membranes.* Journal of Membrane Science, 2001. **188**(1): p. 115-128.
14. Lee, H., G. Amy, J. Cho, Y. Yoon, S.-H. Moon, and I.S. Kim, *Cleaning strategies for flux recovery of an ultrafiltration membrane fouled by natural organic matter.* Water Research, 2001. **35**(14): p. 3301-3308.

15. Wilbert, M.C., *Enhancement of membrane fouing resistance through surface modification*, D.o.t. Interior, Editor. 1997: Denver, CO.
16. Ma, H., C.N. Bowman, and R.H. Davis, *Membrane fouling reduction by backpulsing and surface modification.* Journal of Membrane Science, 2000. **173**(2): p. 191-200.
17. Lim, A.L. and R. Bai, *Membrane fouling and cleaning in microfiltration of activated sludge wastewater.* Journal of Membrane Science, 2003. **216**(1-2): p. 279-290.
18. Kim, A.S. and Y. Liu, *Critical flux of hard sphere suspensions in crossflow filtration: hydrodynamic force bias Monte Carlo simulations.* Journal of Membrane Science, 2008. **323**: p. 67-76.
19. Greengard, L., *The Rapid Evaluation of Potential Fields in Particle Systems*. ACM Distinguished Dissertations. 1988, Cambridge, MA: Massachusetts Institute of Technology.
20. Elimelech, M., J. Gregory, X. Jia, and R.A. Williams, *Particle Deposition & Aggregation: Measurement, Modeling and Simulation*. Colloid and Surface Engineering: Applications in the process industries; Controlled Particle, Droplet and Bubble Formation, ed. R.A. Williams. 1995, Woburn, MA: Butterworth-Heinemann.
21. Hale, J.S., A. Harris, Q. Li, and B.C. Houchens. *The fluid mechanics of membrane filtration.* in *2007 ASME International Mechanical Engineering Congress and Exposition*. 2007. Seattle, WA.
22. Campo, L.M. and B.C. Houchens, *Modeling viscosity variations in membrane fouling*, in *APS Dvivision of Fluid Dynamics 60th Annual Meeting*. 2007: Salt Lake City, UT.
23. Boyd, J.P., *Chebyshev and Fourier Spectral Methods*. Second (Revisesd) ed. 2001, Mineola, NY: Dover Publications.
24. Heath, M.T., *Scientific Computing: An Introductory Survey, Second Edition.* 2002, New York, NY: McGraw-Hill.
25. Parsegan, V.A., *Van der Waals Forces: A Handbook for Biologists, Chemists, Engineers, and Physicists*. 2006, New York, NY: Cambridge University Press.
26. Bhattacharjee, S., A.S. Kim, and M. Elimelech, *Concentration polarization of interacting solute particles in cross-flow membrane filtration.* Journal of Colloid and Interface Science, 1999. **212**: p. 81-99.
27. Kim, A.S. and E.M.V. Hoek, *Cake structure in dead-end membrane filtration: Monte Carlo simulations.* Environmental Engineering Science, 2002. **19**(6): p. 373-386.
28. Li, H., S. Wei, C. Qing, and J. Yang, *Discussion on the position of the shear plane.* Journal of Colloid and Interface Science, 2003. **258**(1): p. 40-44.
29. Chew, W.C. and P.N. Sen, *Potential of a sphere in an ionic solution in thin double layer approximations.* Journal of Chemical Physics, 1982. **77**(4): p. 2042-2044.
30. Hunter, R.J., *Foundations of Colloid Science*. 2001, New York, NY: Oxford University Press.

31. D'evelyn, M.P. and S.A. Rice, *Comment on the configuration space diffusion criterion for optimization of the force bias monte carlo method.* Chemical Physics Letters, 1981. **77**(3): p. 630-633.
32. Rao, M. and B.J. Berne, *On the force-bias monte carlo simulation of simple liquid.* Journal of Chemical Physics, 1979. **71**: p. 129-132.
33. Berman, A.S., *Laminar flow in channels with porous walls.* Journal of Applied Physics, 1953. **24**(9): p. 1232-1235.
34. Happel, J., *Viscous flow in multiparticle systems: slow motion of fluids relative top the beds of spherical particles.* AIChE Journal, 1958. **4**: p. 197-201.
35. Kim, A.S. and Y. Liu, *Irreversible chemical potential and shear-induced diffusion in crossflow filtration.* Industrial Engineering & Chemistry Research, 2008. **47**: p. 5611-5614.
36. Leighton, D. and A. Acrivos, *Viscous resuspension.* Chemical Engineering Science, 1986. **41**(6): p. 1377-1384.
37. Leighton, D. and A. Acrivos, *Measurement of the shear induced coefficient of self-diffusion in concentration suspensions of spheres.* Journal of Fluid Mechanics, 1987. **177**: p. 109-131.
38. Leighton, D. and A. Acrivos, *The shear-induced migration of particles in concentrated suspensions.* Journal of Fluid Mechanics, 1987. **181**: p. 415-439.
39. Gokhale, M., J. Frigo, C. Ahrens, J.L. Tripp, and R. Minnich, *Monte Carlo Radiative Heat Transfer Simulation on a Reconfigurable Computer.* Lecture Notes in Computer Science, 2004. **3203**: p. 95-104.
40. Zhu, X. and M. Elimelech, *Colloidal Fouling of Reverse Osmosis Membranes: Measurements and Fouling Mechanisms.* Environmental Science Technology, 1997. **31**: p. 3654-3662.
41. Kurtz, S., R. Siskey, L. Ciccarelli, A.v. Oooij, J. Peloza, and M. Willarraga, *Retrieval Analysis of Total Disc Replacements: Implications for Standardized Wear Testing.* Journal of ASTM International, 2006. **3**(6).
42. Ihler, A., *An Overview of Fast Multipole Methods*. 2004, Massachusetts Institute of Technology: Cambridge, MA. p. 20.

APPENDIX A: Sample Output

Step	Acceptance ratio	Step Size in Multiples of R	Particles	Order Parameter	Maximum Velocity in Field (m/s)	Wall Time (seconds)
10	0.00E+00	3.47E+00	2100	0.3453	0.7284	4.06
20	3.79E-01	3.30E+00	2100	0.3589	0.7765	7.97
30	4.03E-01	3.13E+00	2100	0.3691	0.8086	11.87
40	4.19E-01	2.98E+00	2100	0.3785	0.8569	15.79
50	4.37E-01	2.83E+00	2100	0.3879	0.8902	19.69
60	4.49E-01	2.69E+00	2100	0.396	0.9171	23.58
70	4.63E-01	2.55E+00	2100	0.4024	0.9385	27.5
80	4.84E-01	2.43E+00	2100	0.4076	0.9781	31.44
90	4.96E-01	2.30E+00	2100	0.4112	0.9948	35.38
100	5.07E-01	2.42E+00	2100	0.4169	1.0056	39.28
110	4.96E-01	2.30E+00	2100	0.4214	1.0292	43.22
120	5.06E-01	2.41E+00	2100	0.4265	1.0512	47.24
130	5.01E-01	2.53E+00	2100	0.4312	1.0638	51.12
140	4.87E-01	2.41E+00	2100	0.436	1.0763	55.03
150	5.02E-01	2.53E+00	2100	0.4402	1.0896	58.93
160	4.91E-01	2.40E+00	2100	0.4448	1.1203	62.81
170	4.99E-01	2.28E+00	2100	0.4489	1.1346	66.73
180	5.17E-01	2.40E+00	2100	0.4525	1.1485	70.63
190	5.01E-01	2.51E+00	2100	0.4551	1.165	74.5
200	4.90E-01	2.39E+00	2100	0.4579	1.1833	78.39
210	4.97E-01	2.27E+00	2100	0.4613	1.1835	82.28
220	5.10E-01	2.38E+00	2100	0.4634	1.1991	86.19
230	4.94E-01	2.26E+00	2100	0.4659	1.2095	90.15
240	5.10E-01	2.38E+00	2100	0.4696	1.2165	94.21
250	5.02E-01	2.50E+00	2100	0.4722	1.2155	98.39
260	4.89E-01	2.37E+00	2100	0.4743	1.228	102.65
270	4.98E-01	2.25E+00	2100	0.4769	1.2419	106.84
280	5.14E-01	2.37E+00	2100	0.4781	1.2506	111.03
290	5.01E-01	2.48E+00	2100	0.4795	1.2604	115.16
300	4.94E-01	2.36E+00	2100	0.4818	1.2633	119.42
310	5.00E-01	2.48E+00	2100	0.4846	1.2709	123.59
320	4.94E-01	2.35E+00	2100	0.4867	1.2744	127.83
330	5.05E-01	2.47E+00	2100	0.4883	1.2779	131.98
340	4.85E-01	2.35E+00	2100	0.4897	1.2801	136.14
350	5.05E-01	2.47E+00	2100	0.4917	1.2806	140.28
360	4.91E-01	2.34E+00	2100	0.4933	1.29	144.43
370	5.04E-01	2.46E+00	2100	0.4956	1.2974	148.59
380	4.97E-01	2.34E+00	2100	0.497	1.3087	152.77
390	4.93E-01	2.22E+00	2100	0.497	1.3069	156.92
400	5.10E-01	2.33E+00	2100	0.4995	1.32	161.09
410	5.02E-01	2.45E+00	2100	0.5003	1.3209	165.24

420	4.85E-01	2.32E+00	2100	0.5019	1.3354	169.39
430	5.01E-01	2.44E+00	2100	0.5031	1.341	173.51
440	4.91E-01	2.32E+00	2100	0.5035	1.339	177.64
450	4.93E-01	2.20E+00	2100	0.5043	1.3439	181.79
460	5.14E-01	2.31E+00	2100	0.5054	1.3477	185.96
470	5.03E-01	2.43E+00	2100	0.5072	1.3502	190.08
480	4.83E-01	2.31E+00	2100	0.5088	1.355	194.23
490	4.87E-01	2.19E+00	2100	0.5097	1.3619	198.37
500	5.11E-01	2.30E+00	2100	0.511	1.3653	202.55
510	5.04E-01	2.42E+00	2100	0.5134	1.3701	206.66
520	4.83E-01	2.30E+00	2100	0.5139	1.3775	210.82
530	4.98E-01	2.18E+00	2100	0.5147	1.38	214.99
540	5.11E-01	2.29E+00	2100	0.5152	1.3746	219.12
550	4.99E-01	2.18E+00	2100	0.5163	1.3804	223.28
560	5.07E-01	2.28E+00	2100	0.5178	1.3877	227.39
570	4.94E-01	2.17E+00	2100	0.5189	1.394	231.57
580	5.10E-01	2.28E+00	2100	0.5196	1.4012	235.69
590	4.99E-01	2.16E+00	2100	0.5194	1.3989	239.88
600	5.07E-01	2.27E+00	2100	0.5204	1.4078	244.04
610	5.02E-01	2.39E+00	2100	0.5208	1.3994	248.13
620	4.82E-01	2.27E+00	2100	0.5215	1.4139	252.33
630	5.06E-01	2.38E+00	2100	0.5218	1.4059	256.45
640	4.80E-01	2.26E+00	2100	0.5231	1.4059	260.57
650	4.92E-01	2.15E+00	2100	0.5221	1.4124	264.74
660	5.18E-01	2.26E+00	2100	0.5224	1.4062	268.87
670	4.94E-01	2.14E+00	2100	0.5227	1.4029	273.05
680	5.05E-01	2.25E+00	2100	0.5234	1.4049	277.17
690	4.99E-01	2.14E+00	2100	0.5228	1.4037	281.37
700	5.12E-01	2.24E+00	2100	0.5243	1.4143	285.51
710	4.90E-01	2.13E+00	2100	0.5258	1.4196	289.64
720	5.16E-01	2.24E+00	2100	0.526	1.4175	293.8
730	5.00E-01	2.13E+00	2100	0.5259	1.4205	297.95
740	5.17E-01	2.23E+00	2100	0.5258	1.4195	302.1
750	5.01E-01	2.34E+00	2100	0.5281	1.4234	306.21
760	4.80E-01	2.23E+00	2100	0.5287	1.4264	310.38
770	4.96E-01	2.12E+00	2100	0.5294	1.4245	314.5
780	5.04E-01	2.22E+00	2100	0.5297	1.4262	318.65
790	4.95E-01	2.11E+00	2100	0.5313	1.4259	322.85
800	5.11E-01	2.22E+00	2100	0.5314	1.4234	327.01
810	5.02E-01	2.33E+00	2100	0.5327	1.429	331.11
820	4.90E-01	2.21E+00	2100	0.5343	1.4357	335.24
830	5.01E-01	2.32E+00	2100	0.5352	1.4337	339.37
840	4.82E-01	2.21E+00	2100	0.5351	1.4373	343.55
850	5.08E-01	2.32E+00	2100	0.5364	1.4453	347.67
860	4.88E-01	2.20E+00	2100	0.537	1.4516	351.82
870	5.08E-01	2.31E+00	2100	0.5365	1.4529	355.93
880	4.86E-01	2.19E+00	2100	0.5371	1.4441	360.07
890	4.94E-01	2.08E+00	2100	0.5379	1.4465	364.22

900	5.27E-01	2.19E+00	2100	0.5386	1.4405	368.37
910	5.01E-01	2.30E+00	2100	0.5384	1.4414	372.48
920	4.80E-01	2.18E+00	2100	0.5397	1.4495	376.62
930	4.98E-01	2.07E+00	2100	0.5394	1.4507	380.78
940	5.16E-01	2.18E+00	2100	0.5398	1.4488	384.92
950	5.02E-01	2.29E+00	2100	0.5401	1.4465	389.06
960	4.88E-01	2.17E+00	2100	0.5404	1.4509	393.22
970	5.14E-01	2.28E+00	2100	0.5412	1.4493	397.36
980	4.90E-01	2.17E+00	2100	0.5416	1.454	401.5
990	5.01E-01	2.28E+00	2100	0.5424	1.4544	405.6
1000	4.84E-01	2.16E+00	2100	0.5427	1.4586	409.75
1010	5.03E-01	2.27E+00	2100	0.5425	1.465	413.86
1020	4.93E-01	2.16E+00	2100	0.5429	1.4616	418
1030	4.99E-01	2.05E+00	2100	0.5429	1.4624	422.17
1040	5.17E-01	2.15E+00	2100	0.5437	1.4617	426.3
1050	5.09E-01	2.26E+00	2100	0.544	1.4569	430.44
1060	4.90E-01	2.15E+00	2100	0.5456	1.4595	434.57
1070	5.05E-01	2.25E+00	2100	0.5458	1.4646	438.78
1080	4.91E-01	2.14E+00	2100	0.5463	1.4704	442.97
1090	4.92E-01	2.03E+00	2100	0.5474	1.4687	447.18
1100	5.15E-01	2.13E+00	2100	0.5468	1.4721	451.34
1110	5.01E-01	2.24E+00	2100	0.5474	1.4865	455.47
1120	4.93E-01	2.13E+00	2100	0.5471	1.4808	459.6
1130	5.06E-01	2.24E+00	2100	0.5478	1.4831	463.73
1140	4.92E-01	2.12E+00	2100	0.5484	1.4699	467.91
1150	5.07E-01	2.23E+00	2100	0.5488	1.4857	472
1160	4.80E-01	2.12E+00	2100	0.5485	1.4846	476.13
1170	5.06E-01	2.22E+00	2100	0.5481	1.4807	480.22
1180	4.92E-01	2.11E+00	2100	0.5488	1.4904	484.5
1190	5.10E-01	2.22E+00	2100	0.5497	1.4847	488.6
1200	4.92E-01	2.11E+00	2100	0.5486	1.4839	492.73
1210	5.00E-01	2.21E+00	2100	0.5495	1.4939	496.86
1220	4.93E-01	2.10E+00	2100	0.5497	1.4911	500.98
1230	5.05E-01	2.21E+00	2100	0.5501	1.483	505.11
1240	4.83E-01	2.10E+00	2100	0.5497	1.484	509.27
1250	4.98E-01	1.99E+00	2100	0.5499	1.4917	513.45
1260	5.18E-01	2.09E+00	2100	0.5498	1.4887	517.59
1270	5.13E-01	2.20E+00	2100	0.5502	1.4861	521.69
1280	4.90E-01	2.09E+00	2100	0.5513	1.4906	525.83
1290	5.04E-01	2.19E+00	2100	0.5519	1.49	529.94
1300	4.90E-01	2.08E+00	2100	0.5525	1.4972	534.05
1310	5.09E-01	2.19E+00	2100	0.5534	1.4934	538.17
1320	4.90E-01	2.08E+00	2100	0.5529	1.4971	542.32
1330	5.08E-01	2.18E+00	2100	0.5532	1.4925	546.41
1340	4.95E-01	2.07E+00	2100	0.5533	1.4901	550.62
1350	5.13E-01	2.18E+00	2100	0.5534	1.4936	554.72
1360	5.01E-01	2.28E+00	2100	0.5528	1.4903	558.85
1370	4.84E-01	2.17E+00	2100	0.5534	1.5014	562.99

1380	5.02E-01	2.28E+00	2100	0.5536	1.5071	567.14
1390	4.74E-01	2.16E+00	2100	0.5532	1.5026	571.23
1400	4.88E-01	2.06E+00	2100	0.5539	1.5103	575.39
1410	5.14E-01	2.16E+00	2100	0.5543	1.5007	579.52
1420	5.03E-01	2.27E+00	2100	0.5541	1.5004	583.6
1430	4.75E-01	2.15E+00	2100	0.5539	1.5091	587.75
1440	4.98E-01	2.05E+00	2100	0.5529	1.512	591.92
1450	5.08E-01	2.15E+00	2100	0.5534	1.4999	596.06
1460	4.92E-01	2.04E+00	2100	0.5543	1.511	600.16
1470	5.05E-01	2.14E+00	2100	0.5549	1.5081	604.31
1480	4.99E-01	2.04E+00	2100	0.5545	1.5108	608.44
1490	5.07E-01	2.14E+00	2100	0.5547	1.5084	612.55
1500	4.93E-01	2.03E+00	2100	0.5545	1.514	616.69
1510	5.07E-01	2.13E+00	2100	0.5531	1.5041	620.81
1520	4.95E-01	2.03E+00	2100	0.5531	1.4932	624.97
1530	5.09E-01	2.13E+00	2100	0.553	1.5003	629.1
1540	4.96E-01	2.02E+00	2100	0.554	1.5095	633.25
1550	5.11E-01	2.12E+00	2100	0.5545	1.5001	637.34
1560	4.97E-01	2.02E+00	2100	0.554	1.5046	641.5
1570	5.13E-01	2.12E+00	2100	0.5538	1.5032	645.65
1580	5.13E-01	2.22E+00	2100	0.5544	1.5017	649.74
1590	4.87E-01	2.11E+00	2100	0.5548	1.5173	653.88
1600	4.98E-01	2.01E+00	2100	0.5549	1.5195	658.04
1610	5.21E-01	2.11E+00	2100	0.5554	1.518	662.17
1620	4.95E-01	2.00E+00	2100	0.5551	1.5085	666.35
1630	5.17E-01	2.10E+00	2100	0.5542	1.5093	670.51
1640	5.04E-01	2.21E+00	2100	0.5539	1.505	674.61
1650	4.90E-01	2.09E+00	2100	0.5536	1.498	678.77
1660	5.02E-01	2.20E+00	2100	0.5552	1.516	682.9
1670	4.87E-01	2.09E+00	2100	0.5558	1.5215	687.01
1680	5.04E-01	2.19E+00	2100	0.5557	1.5117	691.16
1690	4.84E-01	2.08E+00	2100	0.5562	1.5096	695.29
1700	5.11E-01	2.19E+00	2100	0.556	1.5082	699.45
1710	4.96E-01	2.08E+00	2100	0.5567	1.5107	703.61
1720	5.08E-01	2.18E+00	2100	0.5564	1.5133	707.73
1730	4.86E-01	2.07E+00	2100	0.5557	1.5148	711.88
1740	5.05E-01	2.18E+00	2100	0.5562	1.5226	716.03
1750	4.87E-01	2.07E+00	2100	0.5562	1.5174	720.15
1760	5.02E-01	2.17E+00	2100	0.5552	1.5114	724.28
1770	4.94E-01	2.06E+00	2100	0.5569	1.5129	728.41
1780	5.03E-01	2.17E+00	2100	0.5568	1.5159	732.54
1790	4.93E-01	2.06E+00	2100	0.5566	1.5141	736.73
1800	5.10E-01	2.16E+00	2100	0.5576	1.517	740.84
1810	4.97E-01	2.05E+00	2100	0.5578	1.5213	745
1820	5.02E-01	2.16E+00	2100	0.5566	1.5185	749.14
1830	4.97E-01	2.05E+00	2100	0.5558	1.5229	753.3
1840	5.10E-01	2.15E+00	2100	0.5557	1.5171	757.43
1850	4.97E-01	2.04E+00	2100	0.5552	1.5155	761.57

1860	5.04E-01	2.15E+00	2100	0.5548	1.5115	765.69
1870	4.92E-01	2.04E+00	2100	0.5545	1.5119	769.86
1880	5.17E-01	2.14E+00	2100	0.5551	1.5114	773.98
1890	5.00E-01	2.03E+00	2100	0.5544	1.5055	778.13
1900	5.10E-01	2.13E+00	2100	0.5546	1.5	782.26
1910	5.00E-01	2.03E+00	2100	0.5542	1.5083	786.45
1920	5.12E-01	2.13E+00	2100	0.5541	1.5103	790.6
1930	4.96E-01	2.02E+00	2100	0.5542	1.5143	794.78
1940	5.12E-01	2.12E+00	2100	0.5548	1.5094	798.91
1950	4.88E-01	2.02E+00	2100	0.554	1.5131	803.06
1960	5.09E-01	2.12E+00	2100	0.5543	1.5197	807.2
1970	5.02E-01	2.22E+00	2100	0.5542	1.5086	811.29
1980	4.84E-01	2.11E+00	2100	0.5539	1.512	815.44
1990	5.11E-01	2.22E+00	2100	0.5544	1.5126	819.52
2000	4.80E-01	2.11E+00	2100	0.5549	1.5101	823.66

9500	4.95E-01	2.03E+00	2100	0.5698	1.546	3949.98
9510	5.04E-01	2.13E+00	2100	0.5702	1.5447	3954.03
9520	4.91E-01	2.02E+00	2100	0.5706	1.5525	3958.13
9530	5.00E-01	2.13E+00	2100	0.5703	1.5555	3962.2
9540	4.86E-01	2.02E+00	2100	0.5704	1.552	3966.29
9550	4.99E-01	1.92E+00	2100	0.571	1.5515	3970.38
9560	5.13E-01	2.01E+00	2100	0.5709	1.5502	3974.51
9570	5.11E-01	2.12E+00	2100	0.5711	1.5499	3978.57
9580	4.91E-01	2.01E+00	2100	0.5709	1.5556	3982.68
9590	5.15E-01	2.11E+00	2100	0.571	1.5494	3986.76
9600	4.97E-01	2.00E+00	2100	0.5709	1.5547	3990.86
9610	5.04E-01	2.10E+00	2100	0.5708	1.5496	3994.92
9620	4.90E-01	2.00E+00	2100	0.5697	1.5473	3999.02
9630	5.06E-01	2.10E+00	2100	0.5682	1.5412	4003.11
9640	4.95E-01	1.99E+00	2100	0.568	1.5494	4007.2
9650	5.14E-01	2.09E+00	2100	0.5665	1.5372	4011.27
9660	4.92E-01	1.99E+00	2100	0.5663	1.5448	4015.39
9670	5.12E-01	2.09E+00	2100	0.5655	1.5469	4019.49
9680	4.98E-01	1.98E+00	2100	0.5659	1.554	4023.59
9690	5.14E-01	2.08E+00	2100	0.5657	1.5481	4027.69
9700	4.97E-01	1.98E+00	2100	0.566	1.544	4031.77
9710	5.13E-01	2.08E+00	2100	0.5665	1.5507	4035.86
9720	4.98E-01	1.97E+00	2100	0.5664	1.554	4039.95
9730	5.02E-01	2.07E+00	2100	0.5656	1.5488	4044.03
9740	4.97E-01	1.97E+00	2100	0.5655	1.5395	4048.12
9750	5.16E-01	2.07E+00	2100	0.5651	1.5441	4052.21
9760	5.00E-01	1.96E+00	2100	0.5647	1.5386	4056.32
9770	5.21E-01	2.06E+00	2100	0.5644	1.5404	4060.43
9780	5.03E-01	2.17E+00	2100	0.5642	1.5386	4064.5
9790	4.85E-01	2.06E+00	2100	0.5636	1.5319	4068.58
9800	4.97E-01	1.95E+00	2100	0.5625	1.5272	4072.69
9810	5.19E-01	2.05E+00	2100	0.5621	1.529	4076.78

9820	5.03E-01	2.16E+00	2100	0.5622	1.5276	4080.86
9830	4.83E-01	2.05E+00	2100	0.5629	1.5304	4084.98
9840	5.08E-01	2.15E+00	2100	0.5627	1.5257	4089.05
9850	4.94E-01	2.04E+00	2100	0.5626	1.5296	4093.14
9860	4.96E-01	1.94E+00	2100	0.5635	1.5287	4097.26
9870	5.26E-01	2.04E+00	2100	0.563	1.5267	4101.38
9880	5.07E-01	2.14E+00	2100	0.5629	1.5267	4105.46
9890	4.92E-01	2.03E+00	2100	0.5634	1.5212	4109.55
9900	5.03E-01	2.13E+00	2100	0.5629	1.5245	4113.63
9910	4.91E-01	2.03E+00	2100	0.5629	1.5301	4117.72
9920	4.98E-01	1.93E+00	2100	0.5622	1.5318	4121.83
9930	5.21E-01	2.02E+00	2100	0.5608	1.5352	4125.91
9940	5.06E-01	2.12E+00	2100	0.5614	1.5347	4129.98
9950	4.87E-01	2.02E+00	2100	0.5618	1.5365	4134.07
9960	5.04E-01	2.12E+00	2100	0.562	1.5298	4138.15
9970	5.01E-01	2.22E+00	2100	0.562	1.5337	4142.22
9980	4.77E-01	2.11E+00	2100	0.5626	1.5323	4146.29
9990	4.96E-01	2.01E+00	2100	0.5619	1.5337	4150.37
10000	5.07E-01	2.11E+00	2100	0.5619	1.5301	4154.43

APPENDIX B: Code

HS15.f90, Main Function

```
program HSCFMC

! This program performs a Monte Carlo simulation
! in a sectional cubic segment (of crossflow membrane channel) for
! interacting hard spheres with a continuously updating velocity profile.
!
! Periodic boundary conditions are applied in x-(axial) and
! y-(lateral) directions, and impermeable (reflecting) boundary
! condition is used in z-(vertical) direction, in which top and
! bottom exist two permeable membranes.

 IMPLICIT NONE

 !
 ! Use CAPITAL LETTERS for FORTRAN instrinsic command
 ! Use lower case letters for user-oriented computation
 ! Use Mixed Case Letters fro User-Defined Variables and Values.
 ! For Intel FORTRAN compilation use option "-c -names as_is"
 !
 !================================================
 INCLUDE "HSTypes.h"
 INCLUDE "HSParam.h"
 INCLUDE "HSConst.h"
 INCLUDE "/opt/apps/openmpi/1.3.3-intel/include/mpif.h"

 !=== Declaration for General Computation ===
 CHARACTER(19)       :: MyJobTime, MyJobTimeIni
 INTEGER, DIMENSION(3) :: TimeArray
 INTEGER :: idum = 0
 !========== External and Intrinsic Functions ===
 DOUBLE PRECISION    :: RandNum
 DOUBLE PRECISION    :: Tstart, Trunning
 !================================================
 DOUBLE PRECISION    :: RadiusP          ! Radius of Each Particle (Size = 2 *
RadiusP)
 INTEGER :: NPini,  NPfed, NPcompen,   NPfedMin,   NPfedMax
 INTEGER :: istep, ip, ibin, NStart
 INTEGER :: jp, gi, gj
 INTEGER :: iFile, iSeqFile = 0, iSeqFileIni = 0
 DOUBLE PRECISION    :: xbin
 DOUBLE PRECISION    :: Gap
```

```
  LOGICAL :: Overlap, OverlapTest, OverlapCompen

  !================================================
  DOUBLE PRECISION,        DIMENSION(3)          :: BoxSize, SlabSize
  DOUBLE PRECISION                               :: HalfHeight, HalfLength
  DOUBLE PRECISION                               :: Displacement, DisplacementIni
  DOUBLE PRECISION                               :: DisplacementCorrection

  DOUBLE PRECISION,        DIMENSION(:), ALLOCATABLE :: Conc, ConcIni,
ConcAvg, Z, Znorm
  INTEGER,      DIMENSION(:), ALLOCATABLE :: HistConc, HistConcAvg,
HistConcSum

  DOUBLE PRECISION                               :: FlowZnorm

  DOUBLE PRECISION                               :: acatma=0.0D0, acm =0.0D0,
aratio=0.0D0
  DOUBLE PRECISION                               :: FHydroZ , FGravity

  DOUBLE PRECISION                               :: HydroConstZ

  DOUBLE PRECISION                               :: Diff_Brownian
  DOUBLE PRECISION                               :: Diff_Shear_Max
  DOUBLE PRECISION                               :: Diff_Ratio_S2B

  DOUBLE PRECISION                               :: DeltaE
!  DOUBLE PRECISION                               :: DeltaE_SI
  DOUBLE PRECISION                               :: Happel

  DOUBLE PRECISION                               :: VolFracCorrection
  DOUBLE PRECISION                               :: SIcorrection

  DOUBLE PRECISION                               :: ZNormAvg
  DOUBLE PRECISION                               :: local_pxbias
  DOUBLE PRECISION                               :: VolFracAvg
  DOUBLE PRECISION                               :: SResistance

  DOUBLE PRECISION                               :: EngNew, ConcNew
  DOUBLE PRECISION                               :: EngOld, ConcOld

  DOUBLE PRECISION                               :: SUM_Z1 = 0.0D0, SUM_Z2 = 0.0D0,
PSI_NP = 0.0D0, PSI_AVG = 0.0D0

  INTEGER                          :: Nx1
  DOUBLE PRECISION                                        :: DelP, ConcCorrection,
Location
```

```
 DOUBLE PRECISION                                    :: Vel, MaxVel,
MaxVelLoc, Local_Visc, Local_Shear
 DOUBLE PRECISION,      DIMENSION(:), ALLOCATABLE :: x, ConcSlab,
VelCoeff
 DOUBLE PRECISION,            DIMENSION(:), ALLOCATABLE :: ConcCoeff
 DOUBLE PRECISION,            DIMENSION(:,:), ALLOCATABLE :: Tx, Tx1,
Tx2

 DOUBLE PRECISION              :: Move, Acc, DelTime
 DOUBLE PRECISION               :: EnergyOld, EnergyNew, EnergyPart
 LOGICAL                       :: VDWFlag

 INTEGER,      DIMENSION(:), ALLOCATABLE :: Hist
 TYPE(particle), DIMENSION(:), ALLOCATABLE :: Colloid,   ColloidIni, ColloidTri,
ColloidFed, OldColloid
 TYPE(particle)                :: NewColloid, MidColloid, DelColloid

 INTEGER :: stat(MPI_STATUS_SIZE)
 INTEGER :: ParticleType, OldTypes(0:1), BlockCounts(0:1), Offsets(0:1), Extent
 INTEGER                       :: NumTasks, Rank, Dest, Sourse, Count, Tag, ierr
 INTEGER, DIMENSION(:,:), ALLOCATABLE      :: SendBufPart, SendBufPartMem,
SendBufCalc, SendBufCalcMem
 INTEGER, DIMENSION(:), ALLOCATABLE        :: RecvBufPart, RecvBufCalc
 INTEGER, DIMENSION(:,:), ALLOCATABLE       :: BadMat
 INTEGER, DIMENSION(:), ALLOCATABLE         :: BadVec
 INTEGER                                   :: EnergyCalcs
 DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: EnergyVecPart,
ProcEnergyBuf
 DOUBLE PRECISION, DIMENSION(:,:), ALLOCATABLE :: EnergyMat,
EnergyBuf, TotEnergyBuf
 INTEGER                       :: Procs, PartPerProc, CalcPerProc
 INTEGER                            :: Part1, Part2
 DOUBLE PRECISION               :: MaxStepTime, MaxTotalTime
 TYPE(particle), DIMENSION(:), ALLOCATABLE  :: SurfPart
 LOGICAL                            :: SurfTest
 DOUBLE PRECISION                        :: Debye
 DOUBLE PRECISION,      DIMENSION(:,:), ALLOCATABLE :: MaxMoveSize,
SteadyColloid
 DOUBLE PRECISION                :: MoveSize
!  DOUBLE PRECISION                        :: Gam, GamWall, VDWEng, EDLEng,
Debye, Height, OppHeight, eps_0
 !=============================================

CALL MPI_INIT(ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,Rank,ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NumTasks,ierr)
```

```
Procs=NumTasks
IF (MOD(NPmax,NumTasks).NE.0) THEN
  WRITE(*,*) "Number of Particles Must Divide Evenly into Processors"
  CALL MPI_ABORT(MPI_COMM_WORLD,1,ierr)
END IF

!Number of energy calculations to be done in each state evaluation
!Sum from 1 to N-1
EnergyCalcs=NPMax*(NPMax-1)/2
IF (MOD(EnergyCalcs,NumTasks).NE.0) THEN
  WRITE(*,*) "Number of Energy Calculations Must Divide Evenly into Processors"
  CALL MPI_ABORT(MPI_COMM_WORLD,2,ierr)
END IF

Offsets(0)=0
OldTypes(0)=MPI_DOUBLE_PRECISION
BlockCounts(0)=9
CALL MPI_TYPE_EXTENT(MPI_DOUBLE_PRECISION,Extent,ierr)
Offsets(1)=9*Extent
OldTypes(1)=MPI_INTEGER
BlockCounts(1)=3
CALL MPI_TYPE_STRUCT(2,BlockCounts,Offsets,OldTypes,ParticleType,ierr)
CALL MPI_TYPE_COMMIT(ParticleType,ierr)

!WRITE(*,*) "SUCCESSFUL INTIALIZATION!"
CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
IF (Rank.EQ.1) THEN
 ! Initiation of computation
 CALL CPU_TIME (Tstart)
END IF

 RadiusP = 0.5D0 * DiameterP
 VDWCutoff = VDWCutoff / RadiusP

 ! The number of particled to be fed at the epicenter of the channel for concentration
compensation
 VolFracCorrection = VolFracFeed / VolFrac
 NPfed    = INT ( DBLE(NP) * VolFracCorrection / DBLE( 1 + 2 * Nbin ) + 0.5D0 )

 NPfed    = MAX(NPfed,2)

 NPfedMin = INT(DBLE(NPfed) - SQRT(DBLE(NPfed)) + 0.5D0 )
 NPfedMax = INT(DBLE(NPfed) + SQRT(DBLE(NPfed)) + 0.5D0 )

 NPfedMin = NPfed
 NPfedMax = NPfed
```

```
 Nx1 = Nx !REMINANT OF CAMPO'S CODE
 MaxVelLoc=0.0D0

! PRINT *, "NPfedMin, NPfed, NPfedMax = ", NPfedMin, NPfed, NPfedMax

 CALL ITIME(TimeArray) ;
 CALL RANDOM_SEED()

 IF ( InitStructure=="R"  ) THEN

   CALL GETARG(1,MyJobTime)
   MyJobTimeIni = MyJobTime

   ExtraData="Extra.dat"
   PeekData="MC_"//TRIM(MyJobTime)//"_HS_Peek.msg"
   MCRunInf="MC_"//TRIM(MyJobTime)//"_HS_Abstract.msg"
   XYZbegin="MC_"//TRIM(MyJobTime)//"_HS_XYZbegin.xyz"
   XYZfinal="MC_"//TRIM(MyJobTime)//"_HS_XYZfinal.xyz"
   DensityZ="MC_"//TRIM(MyJobTime)//"_HS_DensityZ.dat"
   Velocity="MC_"//TRIM(MyJobTime)//"_HS_Velocity.dat"
   SimulRes="MC_"//TRIM(MyJobTime)//"_HS_SimulRes.dat"

 ELSEIF ( InitStructure=="P"  ) THEN
   iSeqFile = iSeqFilePre
   MyJobTime=Previous(4:18)
   ExtraData="Extra.dat"
   PeekData="MC_"//TRIM(MyJobTime)//"_HS_Peek.msg"
   MCRunInf="MC_"//TRIM(MyJobTime)//"_HS_Abstract.msg"
   XYZbegin="MC_"//TRIM(MyJobTime)//"_HS_XYZbegin.xyz"
   XYZfinal="MC_"//TRIM(MyJobTime)//"_HS_XYZfinal.xyz"
   DensityZ="MC_"//TRIM(MyJobTime)//"_HS_DensityZ.dat"
   Velocity="MC_"//TRIM(MyJobTime)//"_HS_Velocity.dat"
   SimulRes="MC_"//TRIM(MyJobTime)//"_HS_SimulRes.dat"

 END IF


!CHARACTER (LEN=60) ::
Previous="data/20071010_142727/MC_20071010_151757_HS_XYZ00020.xyz"

!   PRINT *, Previous(47:51)
!   PRINT *, Previous(25:39) ; stop

 !=============================================
 !
```

```
 ! Initialization of the Monte Carlo Simulation
 !
 !=============================================

 NPini = NP
 !=== Determining the BoxLength with NP and volume fraction
 !
 CALL InitRecBox   (BoxSize ,NP , NboxX, NboxZ, VolFrac, FixedBox, FixedH)

 Gap = BoxSize(3) / DBLE ( 2 * Nbin + 1 )

 SlabSize    = BoxSize
 SlabSize(3) = Gap
 !PRINT *, BoxSize; PRINT *, SlabSize ; STOP

PartPerProc=NPmax/Procs
CalcPerProc=EnergyCalcs/Procs


 !=== Allocating necessary variables

 ALLOCATE( Colloid(NPmax) , ColloidIni(NPmax) , ColloidTri(NPmax),
OldColloid(NPMax))
 ALLOCATE( ColloidFed(NPfed))

 ALLOCATE( Z   (-Nbin:Nbin) , Znorm   (-Nbin:Nbin) )
 ALLOCATE( Hist      (-Nbin:Nbin) , HistConc   (-Nbin:Nbin))
       Hist = 0              ; HistConc = 0
 ALLOCATE( HistConcAvg(-Nbin:Nbin) , HistConcSum(-Nbin:Nbin))
       HistConcAvg = 0       ; HistConcSum = 0
 ALLOCATE( Conc(-Nbin:Nbin) , ConcIni (-Nbin:Nbin) , ConcAvg(-Nbin:Nbin) )
       Conc = 0.0D0      ; ConcIni = 0.0D0      ; ConcAvg = 0.0D0
 ALLOCATE( x(0:Nx),        Tx(0:Nx,0:Nx1),        Tx1(0:Nx,0:Nx1), &
           Tx2(0:Nx,0:Nx1),   ConcCoeff(1:4*Nx), ConcSlab(0:Nx), &
           VelCoeff(1:Nx+1))
 ALLOCATE( SendBufPart(0:1,0:Procs-1), SendBufPartMem(0:1,0:Procs-1),
RecvBufPart(0:1))
 ALLOCATE( SendBufCalc(0:1,0:Procs-1), SendBufCalcMem(0:1,0:Procs-1),
RecvBufCalc(0:1))
 ALLOCATE( EnergyVecPart(0:CalcPerProc+PartPerProc-1),
EnergyBuf(0:CalcPerProc+PartPerProc-1,0:Procs-1))
 ALLOCATE( TotEnergyBuf(0:PartPerProc-1,0:Procs-1),
ProcEnergyBuf(0:PartPerProc-1))
 ALLOCATE( BadMat(0:PartPerProc-1,0:Procs-1), BadVec(0:PartPerProc-1))
 ALLOCATE( EnergyMat(1:NPMax,1:NPMax+1))
 ALLOCATE( SurfPart(Surf))
```

```
DO ip=0,Procs-1
  SendBufPartMem(0,ip)=ip*PartPerProc+1
  SendBufPartMem(1,ip)=(ip+1)*PartPerProc
  SendBufCalcMem(0,ip)=ip*CalcPerProc+1
  SendBufCalcMem(1,ip)=(ip+1)*CalcPerProc
END DO
SendBufPart=SendBufPartMem
SendBufCalc=0

SendBufCalc(0,0)=1
SendBufCalc(1,0)=2
DO ip=1,Procs-1
        SendBufCalc(0,ip)=SendBufCalc(0,ip-1)
        jp=NPMax-SendBufCalc(1,ip-1)+1
        DO WHILE (jp.LT.(CalcPerProc+1))
                SendBufCalc(0,ip)=SendBufCalc(0,ip)+1
                jp=jp+NPMax-SendBufCalc(0,ip)
        ENDDO
        jp=jp-NPMax+SendBufCalc(0,ip)
        SendBufCalc(1,ip)=SendBufCalc(0,ip)+(CalcPerProc+1-jp)
ENDDO

Displacement    =  ( (Pi / 6.0D0) / VolFrac ) **(1.0D0/3.0D0) * 2.0D0
            ! 2.0 is to generate relaxiation region
DisplacementIni =  Displacement

HalfLength  =   BoxSize(1) / 2.0D0
HalfHeight  =   BoxSize(3) / 2.0D0

FGravity    = - Beta * 4.0D0 * Pi / 3.0D0 * RadiusP**4.0D0 &
          * Gravity * DensityFluid * DeltaSpecificDensity
DO ibin = -Nbin, Nbin
        xbin          = DBLE(ibin)
        Z    (ibin)     = xbin * Gap
        Znorm(ibin)     = Z(ibin) / HalfHeight
END DO
IF (Rank.EQ.1) THEN
  !=== Determining Particle Radius = 1.0 (Monodispersed)
  !
        CALL InitRadius   (Colloid%Radius  ,NPmax)

  !=== Determining Particle Zeta Potential
  !
        CALL InitZetaPot  (Colloid%ZetaPot, NPmax, zeta)
```

```
 !=== Initialize zero time for all particles

        CALL InitTime(Colloid%Time,Colloid%Moves,NPmax)

 !===== Initialize Suface Roughness
        IF (Surf.GT.0) THEN
                IF (InitStructure=="R") THEN
                        CALL InitSurface(SurfPart,Surf,BoxSize,SurfPotential)
                        OPEN(UNIT = 81, FILE = "Surface.dat")
                        DO ip=1,Surf
                                DO jp=1,3
                                        WRITE(81, "(3X,F16.6)", ADVANCE ="NO")
SurfPart(ip)%Coord(jp)
                                ENDDO
                                WRITE(81,*)
                        END DO
                        CLOSE(UNIT = 81)
           ELSEIF (InitStructure=="P") THEN
                        OPEN(UNIT = 81, file="Surface.dat", status="OLD")
                        DO ip=1,Surf
                        READ(81,*) SurfPart(ip)%Coord(1), SurfPart(ip)%Coord(2),
SurfPart(ip)%Coord(3)
                        ENDDO
                        SurfPart%ZetaPot=SurfPotential
                ENDIF
        ENDIF

  !=== Determining Initial Particle Coordinates: Random
  !
        IF ( InitStructure=="R"  ) THEN
                CALL InitCoordRandom    (Colloid, NP, Surf, SurfPart, BoxSize,
VDWCutoff)
                Colloid%InitialZ=Colloid%Coord(3)
!               CALL InitCoordRect (Colloid,NP,BoxSize)
        ELSEIF ( InitStructure=="P"  ) THEN

!    print *, Previous, iSeqFile, MyJobTime

                iFile=16
                OPEN(iFile,file=Previous,status="OLD")
        READ(iFile,*) NP; READ(iFile,*)
        DO ip = 1, NP
                READ(iFile,*) Identity, Colloid(ip)%Coord(1), Colloid(ip)%Coord(2),
Colloid(ip)%Coord(3)
        END Do
        CLOSE(iFile)
```

```
!     PRINT *,  Colloid%Coord(1) ; STOP
      WRITE(*,*) "Successful read from previous!"
         ENDIF

 !=== Storing and printing the initial particle coordinate
 !
         ColloidIni = Colloid
         CALL
PrintColloidCoord(iSeqFile,MyJobTime,NPini,NP,ColloidIni,InitStructure)


 !=== To write down VMD movie information without the final index.
 !=== (Later files will be replaced with newer ones.)
         CALL SetupMovie (BoxSize,Nbin,Gap,MyJobTime,iSeqFileIni,iSeqFile -
1,Grid,InitStructure)

 !=== Double-Check Entire Overlaps
 !  The below is to forcifully check inter-particle overlap(s).
         CALL CheckOverlapInChannel (Colloid ,NP ,BoxSize ,Overlap, .TRUE.,
VDWCutoff)
         IF(Overlap) WRITE(*,*) "Before simulation starts, Inter-Particle Spatial Overlap
observed."
!  stop
END IF

CALL CalculateConc (NP,Colloid,BoxSize,Nbin,Gap,ConcIni,HistConc,Rank)

!===Intialize Chebychev For Velcoity Solution
CALL Cheby(Nx, Nx1, x, Tx, Tx1, Tx2, ConcSlab, HalfHeight)

!===Initialize Pressure Driven Flow
CALL PressDrive(Reynolds,Viscosity,HalfHeight,RadiusP,Density,DelP)
!WRITE(*,*) DelP

CALL
ConcCorrectFactor(BoxSize,NP,VolFrac,ParticleVolPerMass,RadiusP,ConcCorrection)

Debye = SQRT(2.0D0 * 1.0D3 * ElecConc * Avogadro * Electron**2.0D0 *
ElecValence**2.0D0 * Beta / (Eps_0 * Eps_r))

!=============================================================
!=============================================================
!=============================================================
!=============================================================
!=============================================================
!== This file contains information of simulation.
```

```
 INCLUDE "HSinf.f90"
!===========================================
!
! Start of the Monte Carlo Simulation
!
!===========================================

!**************************************************************************
!**************************************************************************
!**************************************************************************
!       OPEN(UNIT = 9, FILE = PeekData)

CALL
MPI_SCATTER(SendBufPart,2,MPI_INTEGER,RecvBufPart,2,MPI_INTEGER,1,MPI_
COMM_WORLD,ierr)
CALL
MPI_SCATTER(SendBufCalc,2,MPI_INTEGER,RecvBufCalc,2,MPI_INTEGER,1,MPI
_COMM_WORLD,ierr)
!CALL MPI_BCAST(Colloid,NPmax,ParticleType,1,MPI_COMM_WORLD,ierr)
!CALL MPI_BCAST(SurfPart,Surf,ParticleType,1,MPI_COMM_WORLD,ierr)
CALL
MPI_BCAST(Colloid%Coord(1),NPMax,MPI_DOUBLE_PRECISION,1,MPI_COMM_
WORLD,ierr)
CALL
MPI_BCAST(Colloid%Coord(2),NPMax,MPI_DOUBLE_PRECISION,1,MPI_COMM_
WORLD,ierr)
CALL
MPI_BCAST(Colloid%Coord(3),NPMax,MPI_DOUBLE_PRECISION,1,MPI_COMM_
WORLD,ierr)
CALL
MPI_BCAST(Colloid%InitialZ,NPMax,MPI_DOUBLE_PRECISION,1,MPI_COMM_
WORLD,ierr)
CALL
MPI_BCAST(Colloid%Radius,NPMax,MPI_DOUBLE_PRECISION,1,MPI_COMM_W
ORLD,ierr)
CALL
MPI_BCAST(Colloid%ZetaPot,NPMax,MPI_DOUBLE_PRECISION,1,MPI_COMM_
WORLD,ierr)
CALL
MPI_BCAST(Colloid%Time,NPMax,MPI_DOUBLE_PRECISION,1,MPI_COMM_W
ORLD,ierr)
CALL
MPI_BCAST(Colloid%OldEnergy,NPMax,MPI_DOUBLE_PRECISION,1,MPI_COM
M_WORLD,ierr)
```

```
CALL
MPI_BCAST(Colloid%NewEnergy,NPMax,MPI_DOUBLE_PRECISION,1,MPI_COM
M_WORLD,ierr)
CALL
MPI_BCAST(Colloid%BadMove,NPMax,MPI_INTEGER,1,MPI_COMM_WORLD,ier
r)
CALL
MPI_BCAST(Colloid%Returned,NPMax,MPI_INTEGER,1,MPI_COMM_WORLD,ierr
)
CALL
MPI_BCAST(Colloid%Moves,NPMax,MPI_INTEGER,1,MPI_COMM_WORLD,ierr)
CALL
MPI_BCAST(SurfPart%Coord(1),Surf,MPI_DOUBLE_PRECISION,1,MPI_COMM_W
ORLD,ierr)
CALL
MPI_BCAST(SurfPart%Coord(2),Surf,MPI_DOUBLE_PRECISION,1,MPI_COMM_W
ORLD,ierr)
CALL
MPI_BCAST(SurfPart%Coord(3),Surf,MPI_DOUBLE_PRECISION,1,MPI_COMM_W
ORLD,ierr)
CALL
MPI_BCAST(SurfPart%InitialZ,Surf,MPI_DOUBLE_PRECISION,1,MPI_COMM_WO
RLD,ierr)
CALL
MPI_BCAST(SurfPart%Radius,Surf,MPI_DOUBLE_PRECISION,1,MPI_COMM_WO
RLD,ierr)
CALL
MPI_BCAST(SurfPart%ZetaPot,Surf,MPI_DOUBLE_PRECISION,1,MPI_COMM_WO
RLD,ierr)
CALL
MPI_BCAST(SurfPart%Time,Surf,MPI_DOUBLE_PRECISION,1,MPI_COMM_WOR
LD,ierr)
CALL
MPI_BCAST(SurfPart%OldEnergy,Surf,MPI_DOUBLE_PRECISION,1,MPI_COMM_
WORLD,ierr)
CALL
MPI_BCAST(SurfPart%NewEnergy,Surf,MPI_DOUBLE_PRECISION,1,MPI_COMM_
WORLD,ierr)
CALL
MPI_BCAST(SurfPart%BadMove,Surf,MPI_INTEGER,1,MPI_COMM_WORLD,ierr)
CALL
MPI_BCAST(SurfPart%Returned,Surf,MPI_INTEGER,1,MPI_COMM_WORLD,ierr)
CALL
MPI_BCAST(SurfPart%Moves,Surf,MPI_INTEGER,1,MPI_COMM_WORLD,ierr)
MaxTotalTime=0.0D0
CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
```

```
IF (InitStructure=="R") THEN
  NStart=-(Nequil-1)
  OPEN(UNIT = 9, FILE = PeekData)
ELSEIF (InitStructure=="P") THEN
  NStart=iSeqFilePre*NMovieInterval+1
  OPEN(UNIT = 9, FILE = PeekData, ACCESS= "APPEND", STATUS = "OLD")
ENDIF
WRITE(*,*) "Successful Initialization"
GeneralMonteCarloLoop: &
DO istep = NStart, Nstep
        CALL CalculateConc    ( NP , Colloid , BoxSize , Nbin , Gap , Conc, HistConc )
!               PRINT *, HistConc ; pause
!==================================================================
!==================================================================
!==================================================================

!=====Calculate velocity profile for each Monte Carlo step
        CALL VelocityCalc(Nx, NP, Colloid%Coord(3), RadiusP, DelP, alpha, eta_0, x,
Tx1, Tx2, &
          ConcSlab, ConcCorrection, BoxSize, MaxVelLoc, MaxVel,
VelCoeff,ConcCoeff)
        IF (Rank.EQ.1) THEN
                SUM_Z1 = 0.0D0
                SUM_Z2 = 0.0D0
!   PrevState = Colloid
                MaxStepTime=0.0D0
        END IF

  ! [A]. Old Colloid
  !
  !===  Saving Colloid as OldColloid and NewColloid
  !
        OldColloid = Colloid


!##########################$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$###############
!*******        CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
!++++++++++++++++++++++++++++Skip Matrix
        Part1=RecvBufCalc(0)
        Part2=RecvBufCalc(1)
        EnergyVecPart=0.0D0
        DO jp=1,CalcPerProc
                CALL
PPEnergy(Colloid,Part1,Part2,BoxSize,HamSS,SS_VDWCutoff,EnergyPart,RadiusP,Eps
_r, &
```

```
Beta,Electron,Valence,ElecConc,VDWFlag,VDWCutoff,SS_EDLCutoff,ElecValence,E
DLChoice,Debye,Rank)
                IF (VDWFlag == .TRUE.) THEN
                        WRITE(*,*) Rank, "Possible  der Waals divergence!3"
                END IF
                EnergyVecPart(Part1)=EnergyVecPart(Part1)+EnergyPart
                EnergyVecPart(Part2)=EnergyVecPart(Part2)+EnergyPart
                Part2=Part2+1
                IF (Part2.GT.NPMax) THEN
                        Part1=Part1+1
                        Part2=Part1+1
                ENDIF
        ENDDO
        DO jp=RecvBufPart(0),RecvBufPart(1)
                CALL
PWEnergy(Colloid(jp),BoxSize,HamPS,EnergyPart,RadiusP,Eps_r, &

Beta,Electron,Valence,ElecConc,WallZeta,WallValence,ElecValence,Debye,Rank)
                EnergyVecPart(jp)=EnergyVecPart(jp)+EnergyPart
                IF (Surf.GT.0) THEN
                        DO ip=1,Surf
                                CALL
PWSEnergy(Colloid(jp),SurfPart(ip),BoxSize,HamPS,SS_VDWCutoff,EnergyPart,Radiu
sP,Eps_r, &

Beta,Electron,Valence,ElecConc,VDWFlag,VDWCutoff,SS_EDLCutoff,SurfValence,El
ecValence,EDLChoice,Debye,Rank)
                                IF (VDWFlag == .TRUE.) THEN
                                        WRITE(*,*) Rank, "Possible van der Waals
divergence!2"
                                END IF
                                EnergyVecPart(jp)=EnergyVecPart(jp)+EnergyPart
                        ENDDO
                ENDIF
        ENDDO
        CALL
MPI_ALLREDUCE(EnergyVecPart,Colloid%OldEnergy,NPMax,MPI_DOUBLE_PRE
CISION,MPI_SUM,MPI_COMM_WORLD,ierr)
!++++++++++++++++++++++++++++Matrix
!       DO jp=1,CalcPerProc
!               CALL
PPEnergy(Colloid,Part1,Part2,BoxSize,HamSS,SS_VDWCutoff,EnergyPart,RadiusP,Eps
_r, &
```

```
!
Beta,Electron,Valence,ElecConc,VDWFlag,VDWCutoff,SS_EDLCutoff,ElecValence,E
DLChoice,Debye,Rank)
!               IF (VDWFlag == .TRUE.) THEN
!                       WRITE(*,*) Rank, "Possible  der Waals divergence!3"
!               END IF
!               EnergyVecPart(jp-1)=EnergyPart
!               Part2=Part2+1
!               IF (Part2.GT.NPMax) THEN
!                       Part1=Part1+1
!                       Part2=Part1+1
!               ENDIF
!       ENDDO
!       DO jp=RecvBufPart(0),RecvBufPart(1)
!               CALL
PWEnergy(Colloid(jp),BoxSize,HamPS,EnergyPart,RadiusP,Eps_r, &
!
Beta,Electron,Valence,ElecConc,WallZeta,WallValence,ElecValence,Debye,Rank)
!               gi=CalcPerProc+jp-RecvBufPart(0)
!               EnergyVecPart(gi)=EnergyPart
!               IF (Surf.GT.0) THEN
!                       DO ip=1,Surf
!                               CALL
PWSEnergy(Colloid(jp),SurfPart(ip),BoxSize,HamPS,SS_VDWCutoff,EnergyPart,Radiu
sP,Eps_r, &
!
Beta,Electron,Valence,ElecConc,VDWFlag,VDWCutoff,SS_EDLCutoff,SurfValence,El
ecValence,EDLChoice,Debye,Rank)
!                               IF (VDWFlag == .TRUE.) THEN
!                                       WRITE(*,*) Rank, "Possible van der Waals
divergence!2"
!                               END IF
!                               EnergyVecPart(gi)=EnergyVecPart(gi)+EnergyPart
!                       ENDDO
!               ENDIF
!       ENDDO
!*******       CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
!++++++++++++Choice 1: Gather all to proccess 1 and then form matrix and perform all
sums on one processor
!       CALL
MPI_GATHER(EnergyVecPart,CalcPerProc+PartPerProc,MPI_DOUBLE_PRECISION,
EnergyBuf, &
!
CalcPerProc+PartPerProc,MPI_DOUBLE_PRECISION,1,MPI_COMM_WORLD,ierr)
!       IF (Rank.EQ.1) THEN
!               Part1=1
```

```
!               Part2=2
!               gi=0
!               gj=0
!               DO jp=1,EnergyCalcs
!                       EnergyMat(Part1,Part2)=EnergyBuf(gi,gj)
!                       EnergyMat(Part2,Part1)=EnergyBuf(gi,gj)
!                       gi=gi+1
!                       Part2=Part2+1
!                       IF (gi.GE.CalcPerProc) THEN
!                               gi=0
!                               gj=gj+1
!                       ENDIF
!                       IF (Part2.GT.NPMax) THEN
!                               Part1=Part1+1
!                               Part2=Part1+1
!                       ENDIF
!               ENDDO
!               gi=0
!               gj=0
!               DO jp=1,NPMax
!                       EnergyMat(jp,NPMax+1)=EnergyBuf(gi+CalcPerProc,gj)
!                       gi=gi+1
!                       IF (gi.GE.PartPerProc) THEN
!                               gi=0
!                               gj=gj+1
!                       ENDIF
!               ENDDO
!               Colloid%OldEnergy=0.0D0
!               DO jp=1,NPMax
!                       DO gi=1,NPMax+1
!
        Colloid(jp)%OldEnergy=Colloid(jp)%OldEnergy+EnergyMat(jp,gi)
!                       ENDDO
!               ENDDO
!       ENDIF
!       CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
!       CALL
MPI_BCAST(Colloid%NewEnergy,NPmax,MPI_DOUBLE_PRECISION,1,MPI_COM
M_WORLD,ierr)
!       CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
!++++++++++Choice 2: Gather all to all and then form full matrix and perform fewer
sums on many processors
!       CALL
MPI_ALLGATHER(EnergyVecPart,CalcPerProc+PartPerProc,MPI_DOUBLE_PRECIS
ION,EnergyBuf, &
```

```
!
CalcPerProc+PartPerProc,MPI_DOUBLE_PRECISION,MPI_COMM_WORLD,ierr)
!       Part1=1
!       Part2=2
!       gi=0
!       gj=0
!       DO jp=1,EnergyCalcs
!               EnergyMat(Part1,Part2)=EnergyBuf(gi,gj)
!               EnergyMat(Part2,Part1)=EnergyBuf(gi,gj)
!               gi=gi+1
!               Part2=Part2+1
!               IF (gi.GE.CalcPerProc) THEN
!                       gi=0
!                       gj=gj+1
!               ENDIF
!               IF (Part2.GT.NPMax) THEN
!                       Part1=Part1+1
!                       Part2=Part1+1
!               ENDIF
!       ENDDO
!       gi=0
!       gj=0
!       DO jp=1,NPMax
!               EnergyMat(jp,NPMax+1)=EnergyBuf(gi+CalcPerProc,gj)
!               gi=gi+1
!               IF (gi.GE.PartPerProc) THEN
!                       gi=0
!                       gj=gj+1
!               ENDIF
!       ENDDO
!       ProcEnergyBuf=0
!       DO jp=0,PartPerProc-1
!               DO gi=1,NPMax+1
!
        ProcEnergyBuf(jp)=ProcEnergyBuf(jp)+EnergyMat(RecvBufPart(0)+jp,gi)
!               ENDDO
!       ENDDO
!*******       CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
!       CALL
MPI_ALLGATHER(ProcEnergyBuf,PartPerProc,MPI_DOUBLE_PRECISION,TotEner
gyBuf, &
!         PartPerProc,MPI_DOUBLE_PRECISION,MPI_COMM_WORLD,ierr)
!       gi=0
!       gj=0
!       DO jp=1,NPMax
!               Colloid(jp)%NewEnergy=TotEnergyBuf(gi,gj)
```

```
!               gi=gi+1
!               IF (gi.GE.PartPerProc) THEN
!                       gi=0
!                       gj=gj+1
!               ENDIF
!       ENDDO
!##########################$$$$$$$$$$$$$$$$$$$$$$$$$$$$$################

!*******        CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
        Colloid%BadMove=0
        Colloid%Returned=0
        IF (Rank.EQ.1) THEN
                ParticleLoop1:&
                DO ip = 1, NP

   ! [B]. New Colloid
   !
   !===  Saving Colloid(ip) as NewColloid

                        NewColloid = Colloid(ip)
   ! [C]. Randomly Updating NewColloid Position
   !
    !Find current particle parameters
                        Location=OldColloid(ip)%Coord(3)/HalfHeight
                        CALL
VelViscShear(Nx1,Location,VelCoeff,Vel,ConcCoeff,eta_0,alpha,&
                          ConcCorrection,Local_Visc,RadiusP,HalfHeight,Local_Shear,x)
                        local_pxbias = Vel/MaxVel
!                        local_pxbias = 0.0D0
                        DisplacementCorrection = ( 0.1D0  +  local_pxbias )
                        CALL RANDOM_NUMBER(RandNum)
                        DelColloid%Coord(1) = (2.0D0*RandNum-1.0D0+local_pxbias) *
Displacement * &
                          DisplacementCorrection
                        CALL RANDOM_NUMBER(RandNum)
                        DelColloid%Coord(2) = (2.0D0*RandNum-1.0D0) *
Displacement * DisplacementCorrection
                        CALL RANDOM_NUMBER(RandNum)
                        DelColloid%Coord(3) = (2.0D0*RandNum-1.0D0) *
Displacement * DisplacementCorrection

                        NewColloid%Coord =   OldColloid(ip)%Coord  +
DelColloid%Coord
                        MidColloid%Coord = ( OldColloid(ip)%Coord  +
NewColloid%Coord ) / 2.0D0
```

```
   ! [D].1. Periodic Boundary Conditions in x- and y-directions
   !
                        NewColloid%Coord(1) = NewColloid%Coord(1) - &
                          ANINT(NewColloid%Coord(1)/BoxSize(1))*BoxSize(1)
                        NewColloid%Coord(2) = NewColloid%Coord(2) - &
                          ANINT(NewColloid%Coord(2)/BoxSize(2))*BoxSize(2)
!                        NewColloid%Coord(3) = NewColloid%Coord(3) - &
!                         ANINT(NewColloid%Coord(3)/BoxSize(3))*BoxSize(3)

   ! [D].2. Projective-Reflecting Boundary Conditions in z-direction
                        SurfTest=.FALSE.
                        IF (Surf.GT.0) THEN
                              DO jp=1,Surf
                                    CALL
CheckOverlapIJPer(Colloid(ip),SurfPart(jp),BoxSize,SurfTest,VDWCutoff)
                                    IF(SurfTest==.TRUE.) EXIT
                              ENDDO
                        ENDIF
                        IF    ((NewColloid%Coord(3) >  (HalfHeight    - 1.0D0 *
NewColloid%Radius)).OR.((SurfTest==.TRUE.).AND.(NewColloid%Coord(3).GT.0)) )
THEN
                              NewColloid%Coord(3) =   OldColloid(ip)%Coord(3)
!                              NewColloid%Coord(2) =   OldColloid(ip)%Coord(2)
!                              NewColloid%Coord(1) =   OldColloid(ip)%Coord(1)
                        ELSEIF((NewColloid%Coord(3) <  -(HalfHeight    - 1.0D0 *
NewColloid%Radius)).OR.((SurfTest==.TRUE.).AND.(NewColloid%Coord(3).LT.0)) )
THEN
                              NewColloid%Coord(3) =   OldColloid(ip)%Coord(3)
!                              NewColloid%Coord(2) =   OldColloid(ip)%Coord(2)
!                              NewColloid%Coord(1) =   OldColloid(ip)%Coord(1)
                        END IF


   ! [E]. Temporarily Replacing Colloid(ip) by NewColloid and Checking Overlap
   !
   !=== Calculate Engery_Overlp_ConcGrad of NewColloid

                        Colloid(ip) = NewColloid

                        CALL
Calculate_Overlap_Energy_Conc_Grad(ip,Colloid,NP,BoxSize,Overlap,.FALSE., &
                          EngNew,ConcNew,VDWCutoff)
                        SurfTest=.FALSE.
                        IF (Surf.GT.0) THEN
                              DO jp=1,Surf
```

```
                                        CALL
CheckOverlapIJPer(Colloid(ip),SurfPart(jp),BoxSize,SurfTest,VDWCutoff)
                                        IF(SurfTest==.TRUE.) EXIT
                                    ENDDO
                                ENDIF
                                IF    ((Overlap==.TRUE.).OR.(SurfTest==.TRUE.)) THEN
                                Colloid(ip)%Coord = OldColloid(ip)%Coord
                                    Colloid(ip)%BadMove=1
                                    Colloid(ip)%Returned=1
                                ENDIF
                            ENDDO &
                            ParticleLoop1
                    ENDIF
!*******       CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
                    CALL
MPI_BCAST(Colloid%Coord(1),NPMax,MPI_DOUBLE_PRECISION,1,MPI_COMM_
WORLD,ierr)
                    CALL
MPI_BCAST(Colloid%Coord(2),NPMax,MPI_DOUBLE_PRECISION,1,MPI_COMM_
WORLD,ierr)
                    CALL
MPI_BCAST(Colloid%Coord(3),NPMax,MPI_DOUBLE_PRECISION,1,MPI_COMM_
WORLD,ierr)
                    CALL
MPI_BCAST(Colloid%BadMove,NPMax,MPI_INTEGER,1,MPI_COMM_WORLD,ier
r)
                    CALL
MPI_BCAST(Colloid%Returned,NPMax,MPI_INTEGER,1,MPI_COMM_WORLD,ierr
)
!++++++++++++++++++++More or less effient that broadcasting the entire colloid
map?
!##########################$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
!*******       CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
!++++++++++++++++++++++++++++++Skip Matrix
                    Part1=RecvBufCalc(0)
                    Part2=RecvBufCalc(1)
                    EnergyVecPart=0.0D0
                    DO jp=1,CalcPerProc
                        CALL
PPEnergy(Colloid,Part1,Part2,BoxSize,HamSS,SS_VDWCutoff,EnergyPart,RadiusP,Eps
_r, &

Beta,Electron,Valence,ElecConc,VDWFlag,VDWCutoff,SS_EDLCutoff,ElecValence,E
DLChoice,Debye,Rank)
                        IF (VDWFlag == .TRUE.) THEN
                            WRITE(*,*) Rank, "Possible  der Waals divergence!3"
```

```
                END IF
                EnergyVecPart(Part1)=EnergyVecPart(Part1)+EnergyPart
                EnergyVecPart(Part2)=EnergyVecPart(Part2)+EnergyPart
                Part2=Part2+1
                IF (Part2.GT.NPMax) THEN
                        Part1=Part1+1
                        Part2=Part1+1
                ENDIF
        ENDDO
        DO jp=RecvBufPart(0),RecvBufPart(1)
                CALL
PWEnergy(Colloid(jp),BoxSize,HamPS,EnergyPart,RadiusP,Eps_r, &

Beta,Electron,Valence,ElecConc,WallZeta,WallValence,ElecValence,Debye,Rank)
                EnergyVecPart(jp)=EnergyVecPart(jp)+EnergyPart
                IF (Surf.GT.0) THEN
                        DO ip=1,Surf
                                CALL
PWSEnergy(Colloid(jp),SurfPart(ip),BoxSize,HamPS,SS_VDWCutoff,EnergyPart,Radiu
sP,Eps_r, &

Beta,Electron,Valence,ElecConc,VDWFlag,VDWCutoff,SS_EDLCutoff,SurfValence,El
ecValence,EDLChoice,Debye,Rank)
                                IF (VDWFlag == .TRUE.) THEN
                                        WRITE(*,*) Rank, "Possible van der Waals
divergence!2"
                                END IF
                                EnergyVecPart(jp)=EnergyVecPart(jp)+EnergyPart
                        ENDDO
                ENDIF
        ENDDO
        CALL
MPI_ALLREDUCE(EnergyVecPart,Colloid%NewEnergy,NPMax,MPI_DOUBLE_PRE
CISION,MPI_SUM,MPI_COMM_WORLD,ierr)
!++++++++++++++++++++++++++++Matrix
!       DO jp=1,CalcPerProc
!               CALL
PPEnergy(Colloid,Part1,Part2,BoxSize,HamSS,SS_VDWCutoff,EnergyPart,RadiusP,Eps
_r, &
!
Beta,Electron,Valence,ElecConc,VDWFlag,VDWCutoff,SS_EDLCutoff,ElecValence,E
DLChoice,Debye,Rank)
!               IF (VDWFlag == .TRUE.) THEN
!                       WRITE(*,*) Rank, "Possible  der Waals divergence!3"
!               END IF
!               EnergyVecPart(jp-1)=EnergyPart
```

```
!               Part2=Part2+1
!               IF (Part2.GT.NPMax) THEN
!                       Part1=Part1+1
!                       Part2=Part1+1
!               ENDIF
!       ENDDO
!       DO jp=RecvBufPart(0),RecvBufPart(1)
!               CALL
PWEnergy(Colloid(jp),BoxSize,HamPS,EnergyPart,RadiusP,Eps_r, &
!
Beta,Electron,Valence,ElecConc,WallZeta,WallValence,ElecValence,Debye,Rank)
!               gi=CalcPerProc+jp-RecvBufPart(0)
!               EnergyVecPart(gi)=EnergyPart
!               IF (Surf.GT.0) THEN
!                       DO ip=1,Surf
!                               CALL
PWSEnergy(Colloid(jp),SurfPart(ip),BoxSize,HamPS,SS_VDWCutoff,EnergyPart,Radiu
sP,Eps_r, &
!
Beta,Electron,Valence,ElecConc,VDWFlag,VDWCutoff,SS_EDLCutoff,SurfValence,El
ecValence,EDLChoice,Debye,Rank)
!                               IF (VDWFlag == .TRUE.) THEN
!                                       WRITE(*,*) Rank, "Possible van der Waals
divergence!2"
!                               END IF
!                               EnergyVecPart(gi)=EnergyVecPart(gi)+EnergyPart
!                       ENDDO
!               ENDIF
!       ENDDO
!*******       CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
!++++++++++Choice 1: Gather all to proccess 1 and then form matrix and perform all
sums on one processor
!       CALL
MPI_GATHER(EnergyVecPart,CalcPerProc+PartPerProc,MPI_DOUBLE_PRECISION,
EnergyBuf, &
!
CalcPerProc+PartPerProc,MPI_DOUBLE_PRECISION,1,MPI_COMM_WORLD,ierr)
!       IF (Rank.EQ.1) THEN
!               Part1=1
!               Part2=2
!               gi=0
!               gj=0
!               DO jp=1,EnergyCalcs
!                       EnergyMat(Part1,Part2)=EnergyBuf(gi,gj)
!                       EnergyMat(Part2,Part1)=EnergyBuf(gi,gj)
!                       gi=gi+1
```

```
!                       Part2=Part2+1
!                       IF (gi.GE.CalcPerProc) THEN
!                               gi=0
!                               gj=gj+1
!                       ENDIF
!                       IF (Part2.GT.NPMax) THEN
!                               Part1=Part1+1
!                               Part2=Part1+1
!                       ENDIF
!               ENDDO
!               gi=0
!               gj=0
!               DO jp=1,NPMax
!                       EnergyMat(jp,NPMax+1)=EnergyBuf(gi+CalcPerProc,gj)
!                       gi=gi+1
!                       IF (gi.GE.PartPerProc) THEN
!                               gi=0
!                               gj=gj+1
!                       ENDIF
!               ENDDO
!               Colloid%OldEnergy=0.0D0
!               DO jp=1,NPMax
!                       DO gi=1,NPMax+1
!
        Colloid(jp)%OldEnergy=Colloid(jp)%OldEnergy+EnergyMat(jp,gi)
!                       ENDDO
!               ENDDO
!       ENDIF
!       CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
!       CALL
MPI_BCAST(Colloid%NewEnergy,NPmax,MPI_DOUBLE_PRECISION,1,MPI_COM
M_WORLD,ierr)
!       CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
!++++++++++Choice 2: Gather all to all and then form full matrix and perform fewer
sums on many processors
!       CALL
MPI_ALLGATHER(EnergyVecPart,CalcPerProc+PartPerProc,MPI_DOUBLE_PRECIS
ION,EnergyBuf, &
!
CalcPerProc+PartPerProc,MPI_DOUBLE_PRECISION,MPI_COMM_WORLD,ierr)
!       Part1=1
!       Part2=2
!       gi=0
!       gj=0
!       DO jp=1,EnergyCalcs
!               EnergyMat(Part1,Part2)=EnergyBuf(gi,gj)
```

```
!               EnergyMat(Part2,Part1)=EnergyBuf(gi,gj)
!               gi=gi+1
!               Part2=Part2+1
!               IF (gi.GE.CalcPerProc) THEN
!                       gi=0
!                       gj=gj+1
!               ENDIF
!               IF (Part2.GT.NPMax) THEN
!                       Part1=Part1+1
!                       Part2=Part1+1
!               ENDIF
!       ENDDO
!       gi=0
!       gj=0
!       DO jp=1,NPMax
!               EnergyMat(jp,NPMax+1)=EnergyBuf(gi+CalcPerProc,gj)
!               gi=gi+1
!               IF (gi.GE.PartPerProc) THEN
!                       gi=0
!                       gj=gj+1
!               ENDIF
!       ENDDO
!       ProcEnergyBuf=0
!       DO jp=0,PartPerProc-1
!               DO gi=1,NPMax+1
!
        ProcEnergyBuf(jp)=ProcEnergyBuf(jp)+EnergyMat(RecvBufPart(0)+jp,gi)
!               ENDDO
!       ENDDO
!*******       CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
!       CALL
MPI_ALLGATHER(ProcEnergyBuf,PartPerProc,MPI_DOUBLE_PRECISION,TotEner
gyBuf, &
!        PartPerProc,MPI_DOUBLE_PRECISION,MPI_COMM_WORLD,ierr)
!       gi=0
!       gj=0
!       DO jp=1,NPMax
!               Colloid(jp)%NewEnergy=TotEnergyBuf(gi,gj)
!               gi=gi+1
!               IF (gi.GE.PartPerProc) THEN
!                       gi=0
!                       gj=gj+1
!               ENDIF
!       ENDDO
!#########################$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
#####################$$$$$$$$$$$$$$$$$$$$$$
```

```
!*******       CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
        ParticleLoop2:&
        DO jp=0,PartPerProc-1
                ip=jp+RecvBufPart(0)
                Location=OldColloid(ip)%Coord(3)/HalfHeight
                CALL
VelViscShear(Nx1,Location,VelCoeff,Vel,ConcCoeff,eta_0,alpha,&
                        ConcCorrection,Local_Visc,RadiusP,HalfHeight,Local_Shear,x)
        CALL Calculate_Overlap_Energy_Conc_Grad
(ip,OldColloid,NP,BoxSize,Overlap,.FALSE.,EngOld,ConcOld,VDWCutoff)
                IF (Colloid(ip)%BadMove.EQ.0) THEN
!=IFistep==IFistep==IFistep==IFistep==IFistep=IFistep===IFistep==IFistep==IFistep=
                        IFistep: &
                        IF(istep > 0) THEN

!========= Calculating Average Z-coordinate and volume fraction ===========

                        ZNormAvg   = ( OldColloid(ip)%Coord(3) ) / HalfHeight
                VolFracAvg =   ConcOld

!=============== Shear-Induced Diffusive Force ==============

!               DeltaE_SI =   Diff_Ratio_S2B *  ABS( ZNormAvg ) &
!                * ( SIcorrection (VolFracAvg) / VolFracAvg ) * Happel(VolFracAvg) &
!                * ( ConcNew - ConcOld  )

!=============== Hydrodynamic Bias Force ===============
!               ( HydroConstZ  =   Beta * 6.0D0 * Pi * Viscosity * RadiusP**2 *
Permeate )

!                               WRITE(*,*) Local_Visc
                        HydroConstZ =   Beta * 6.0D0 * Pi * Local_Visc *
RadiusP**2.0D0 * Permeate

                        Diff_Brownian  = (1.0D0/Beta) / ( 6.0D0 * Pi * Local_Visc *
RadiusP )
                        Diff_Shear_Max = ABS(Local_Shear) * RadiusP**2.0D0
                        Diff_Ratio_S2B = Diff_Shear_Max / Diff_Brownian

                FlowZnorm =   0.5D0 * ZNormAvg * ( 3.0D0 - ZNormAvg * ZNormAvg
)

                FHydroZ   =   HydroConstZ * FlowZnorm* Happel(VolFracAvg) / &
                 ( 1.0D0 + Diff_Ratio_S2B * SIcorrection (VolFracAvg) )
```

```
                DeltaE    =   - Lambda * ( FHydroZ + FGravity ) *
(Colloid(ip)%Coord(3)-OldColloid(ip)%Coord(3)) + &
                        Beta * (Colloid(ip)%NewEnergy - Colloid(ip)%OldEnergy)

                ELSEIF(istep <= 0 ) THEN

                        DeltaE  = 0.0D0

                END IF &
                IFistep
!=IFistep==IFistep==IFistep==IFistep==IFistep=IFistep===IFistep==IFistep==IFistep=


!=IFDeltaE==IFDeltaE==IFDeltaE==IFDeltaE==IFDeltaE==IFDeltaE==IFDeltaE=
                IFDeltaE: &
                IF(DeltaE <= 0.0D0) THEN
                        BadVec(jp) = 0

                ELSEIF (DeltaE > 0.0D0) THEN
                                CALL RANDOM_NUMBER(RandNum)
                        IF(EXP(-DeltaE) > RandNum ) THEN
                                        BadVec(jp) = 0

                                ELSE
                                        BadVec(jp) = 1
                                END IF
                        END IF &
                IFDeltaE
!=IFDeltaE==IFDeltaE==IFDeltaE==IFDeltaE==IFDeltaE==IFDeltaE==IFDeltaE=
                ELSE
                        BadVec(jp) = 1
                ENDIF
        ENDDO &
        ParticleLoop2
!*******       CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
        CALL
MPI_GATHER(BadVec,PartPerProc,MPI_INTEGER,BadMat,PartPerProc,MPI_INTEG
ER,1,MPI_COMM_WORLD,ierr)
        IF (Rank.EQ.1) THEN
                gi=0
                gj=0
                DO jp=1,NPMax
                        Colloid(jp)%BadMove=BadMat(gi,gj)
                        gi=gi+1
                        IF (gi.GE.PartPerProc) THEN
                                gi=0
```

```
                        gj=gj+1
                    ENDIF
                ENDDO
!CASCADE
                DO ip=1,NPMax
                    IF
((Colloid(ip)%BadMove.EQ.1).AND.(Colloid(ip)%Returned.EQ.0)) THEN
                        Colloid(ip)%Coord=OldColloid(ip)%Coord
                        Colloid(ip)%Returned=1
                        CALL
Cascade(Colloid,OldColloid,ip,NP,BoxSize,VDWCutoff)
                    ENDIF
                ENDDO
!END CASCADE
                DO ip=1,NPMax
                    acm=acm+1.0D0
                    SUM_Z1 = SUM_Z1 +   Colloid(ip)%Coord(3) / HalfHeight
                    SUM_Z2 = SUM_Z2 + ( Colloid(ip)%Coord(3) / HalfHeight
)**2.0D0
                    IF (Colloid(ip)%Returned.EQ.0) THEN
                        acatma=acatma+1.0D0
                        Colloid(ip)%Moves=Colloid(ip)%Moves+1
                    ENDIF
                ENDDO
        ENDIF
!*******        CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
        CALL
MPI_BCAST(Colloid%Coord(1),NPMax,MPI_DOUBLE_PRECISION,1,MPI_COMM_
WORLD,ierr)
        CALL
MPI_BCAST(Colloid%Coord(2),NPMax,MPI_DOUBLE_PRECISION,1,MPI_COMM_
WORLD,ierr)
        CALL
MPI_BCAST(Colloid%Coord(3),NPMax,MPI_DOUBLE_PRECISION,1,MPI_COMM_
WORLD,ierr)
        CALL
MPI_BCAST(Colloid%Moves,NPMax,MPI_INTEGER,1,MPI_COMM_WORLD,ierr)
!*******        CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
!++++++++++++++++++++More or less effient that broadcasting the entire colloid
map?
!++++++++++++++++++++CURRENT MODS END

!=================================================================
================================
!=================================================================
================================
```

```
!=================================================================
================================

      IF (Rank.EQ.1) THEN
            IF (istep.EQ.(Nstep-EndSteps)) THEN
                  DO ip=1,NPMax
                        SteadyColloid(ip,1)=Colloid(ip)%Coord(1)
                        SteadyColloid(ip,2)=Colloid(ip)%Coord(2)
                        SteadyColloid(ip,3)=Colloid(ip)%Coord(3)
                        MaxMoveSize(ip,1)=ABS(OldColloid(ip)%Coord(1)-
Colloid(ip)%Coord(1))
                        IF (MaxMoveSize(ip,1).GT.Displacement*1.1*2.1) THEN
                              MaxMoveSize(ip,1)=BoxSize(1)-
MaxMoveSize(ip,1)
                        ENDIF
                        MaxMoveSize(ip,2)=ABS(OldColloid(ip)%Coord(2)-
Colloid(ip)%Coord(2))
                        IF (MaxMoveSize(ip,2).GT.Displacement*1.1) THEN
                              MaxMoveSize(ip,2)=BoxSize(2)-
MaxMoveSize(ip,2)
                        ENDIF
                        MaxMoveSize(ip,3)=ABS(OldColloid(ip)%Coord(3)-
Colloid(ip)%Coord(3))
                  ENDDO
            ENDIF
            IF (istep.GT.(Nstep-EndSteps)) THEN
                  DO ip=1,NPMax
                        MoveSize=ABS(OldColloid(ip)%Coord(1)-
Colloid(ip)%Coord(1))
                        IF (MoveSize.GT.Displacement*1.1*2.1) THEN
                              MoveSize=BoxSize(1)-MoveSize
                              IF (MoveSize.LT.0) THEN
                                    WRITE(*,*) "Movement Sizing Problem!2"
                              ENDIF
                        ENDIF
                        IF (MoveSize.GT.MaxMoveSize(ip,1)) THEN
                              MaxMoveSize(ip,1)=MoveSize
                        ENDIF
                        MoveSize=ABS(OldColloid(ip)%Coord(2)-
Colloid(ip)%Coord(2))
                        IF (MoveSize.GT.Displacement*1.1) THEN
                              MoveSize=BoxSize(2)-MoveSize
                              IF (MoveSize.LT.0) THEN
                                    WRITE(*,*) "Movement Sizing Problem!2"
                              ENDIF
                        ENDIF
```

```
                                IF (MoveSize.GT.MaxMoveSize(ip,2)) THEN
                                        MaxMoveSize(ip,2)=MoveSize
                                ENDIF
                                MoveSize=ABS(OldColloid(ip)%Coord(3)-
Colloid(ip)%Coord(3))
                                IF (MoveSize.GT.MaxMoveSize(ip,3)) THEN
                                        MaxMoveSize(ip,3)=MoveSize
                                ENDIF
                        ENDDO
                ENDIF
                PSI_NP = SUM_Z2 / DBLE(NP) - ( SUM_Z1 / DBLE(NP))**2.0D0

!                PRINT *, PSI_NP ; pause

                IF(istep > Nstep/NDdataCollect ) THEN
                PSI_AVG = PSI_AVG + PSI_NP
                END IF

 !=== To monitor the simulation status using "bpeek" command intermittently
 !
                IF(MOD(istep,Npeek)==0) THEN
                CALL CPU_TIME (Trunning)
                WRITE &
                        (*,'(I5," ( out of ",I5,") ",2(2X,E12.6), 2X, I5, 2X, F12.4, 2X,
F12.4, 2X, F12.2)') &
                 istep,Nstep,aratio,Displacement,NP,PSI_NP,MaxVel,Trunning-Tstart
                WRITE &
                        (9,'(I5," ( out of ",I5,") ",2(2X,E12.6), 2X, I5, 2X, F12.4, 2X,
F12.4, 2X, F12.2)') &
                 istep,Nstep,aratio,Displacement,NP,PSI_NP,MaxVel,Trunning-Tstart
                END IF

 !=== To store xyz coordinates periodically to generate a mpeg movie using VMD
 !
                IF(istep <= NmovieMaxStep .AND. MOD(istep,NMovieInterval)==0)
THEN
!                Print *, iSeqFile,MyJobTime,NPini,NP,Colloid
                CALL
PrintColloidCoord(iSeqFile,MyJobTime,NPini,NP,Colloid,InitStructure)
                CALL  SetupMovie (BoxSize,Nbin,Gap,MyJobTime,iSeqFileIni,iSeqFile
- 1,Grid,InitStructure)
                NMovieInterval = INT( DBLE(NMovieInterval) * 1.0001D0 )
                END IF

 !=== To continually update the maximum displacement
 !
```

```
                IF(MOD(istep,Nupdate)==0) THEN
                CALL UpdateDisplacement  (acatma, acm, aratio, Displacement )
                IF ( Displacement > (DisplacementIni) ) Displacement = DisplacementIni
                IF ( Displacement < (DisRestart) ) Displacement = DisplacementIni

                IF(istep > Nstep/NDdataCollect ) THEN

                        CALL UpdateHistogram    (NP,Colloid,BoxSize,Gap,Hist,Nbin)
                        ConcAvg = DBLE(Hist)/(DBLE(istep-Nstep/2)/DBLE(Nupdate))
&
                         *(4.0D0/3.0D0)*Pi/(BoxSize(1)*BoxSize(2)*Gap)
                        iFile=13
                        OPEN(iFile,file=DensityZ,status="REPLACE")
                        DO ibin = -Nbin, Nbin

        WRITE(iFile,'(2X,E12.10,2X,E12.10,2X,E12.10,2X,E12.10)') Znorm(ibin) ,
ConcIni(ibin) , &
                                        Conc(ibin) , ConcAvg(ibin)
                        END Do
                        CLOSE(iFile)

                END IF

                END IF

!9999999999999999999999999999999999999999999999999999999999999999999999999
!9999999999999999999999999999999999999999999999999999999999999999999999999
!9999999999999999999999999999999999999999999999999999999999999999999999999
!9999999999999999999999999999999999999999999999999999999999999999999999999
!9999999999999999999999999999999999999999999999999999999999999999999999999
 !=== To compensate particles near the epicenter to keep the feed volume fraction
 !

!GOTO 8888

                HistConcSum = HistConcSum + HistConc

                IF(istep > 0 .AND. MOD(istep,NupdateCK)==0) THEN

                        HistConcAvg = HistConcSum / NupdateCK   ;    HistConcSum = 0
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++

!=======================================
                InsertSectionBySection :&
                DO ibin = -Nbin, Nbin
```

```
                    Compensation : &
                    IF( HistConcAvg(ibin) < NPfedMin ) THEN
                    ! Forbidding volume expansion
                    ! VolumeExpansion = "N"
                    ! Calculate how many particles will be compensated.
                            NPcompen =  NPfedMax ! -  HistConcAvg(ibin)
                            NotToExceedNPmax : &
                            IF( NP + NPcompen < NPmax ) THEN
                                    OverlapCompen = .TRUE.
                                    DO WHILE (OverlapCompen)
                                    CALL InsertCoordRandom  (ColloidFed        ,
NPcompen , SlabSize , VDWCutoff)
                                    ColloidFed%Coord(3)   =  ColloidFed%Coord(3) +
DBLE(ibin)*SlabSize(3)
                                    CALL InitRadius       (ColloidFed%Radius  ,
NPcompen          )
                                    CALL InitZetaPot      (ColloidFed%ZetaPot ,
NPcompen , zeta     )

                                    OverlapTest = .FALSE.
                                    DO jp = 1, NPcompen
                                            Colloid(NP+jp) = ColloidFed(jp)
                                            CALL
Calculate_Overlap_Energy_Conc_Grad &

(NP+jp,Colloid,NP+jp,BoxSize,Overlap,.FALSE.,EngNew,ConcNew,VDWCutoff)
                                            IF(Overlap.eq..TRUE.) OverlapTest =
Overlap
                                    END DO
                                    OverlapCompen = OverlapTest
                                    END DO
                                    NP = NP + NPcompen
!                                    CALL
PrintColloidCoord(iSeqFile,MyJobTime,NPini,NP,Colloid,InitStructure)
                            END IF &
                            NotToExceedNPmax
                    END IF &
                    Compensation
            END DO &
            InsertSectionBySection
!           PRINT *, NP, NPfed
!           PRINT *, HistConcAvg ; pause
!           CALL
PrintColloidCoord(iSeqFile,MyJobTime,NPini,NP,Colloid,InitStructure)

!=======================================
```

```
!=========================================
!=========================================
!      IF( istep > Nstep*(NDdataCollect-1)/NDdataCollect .AND. VolumeExpansion ==
"Y") THEN
!        IF( ibin == 0 ) THEN
!          VolumeExpansionTest : &
!          IF( HistConcAvg(0)  >  NPfed ) THEN
!             BoxSize(1) = BoxSize(1)*1.1D0
!             BoxSize(2) = BoxSize(2)*1.1D0
!              NP = NP - (HistConcAvg(0)  -  NPfed )
!          END IF &
!          VolumeExpansionTest
!        END IF
!      END IF
!=========================================
!=========================================
!!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++

                  END IF
          END IF
!8888 CONTINUE

!99999999999999999999999999999999999999999999999999999999999999999999999999
!99999999999999999999999999999999999999999999999999999999999999999999999999
!99999999999999999999999999999999999999999999999999999999999999999999999999
!99999999999999999999999999999999999999999999999999999999999999999999999999
!99999999999999999999999999999999999999999999999999999999999999999999999999
END DO &
GeneralMonteCarloLoopCALL MPI_BARRIER(MPI_COMM_WORLD,ierr)

IF (Rank.EQ.1) THEN
        CLOSE(UNIT = 9)
!**************************************************************************
!**************************************************************************
!**************************************************************************

 ! Checking the Entire Overlap among All the Particles after the Simulation
 !
        CALL CheckOverlapInChannel (Colloid ,NP ,BoxSize ,Overlap,
.TRUE.,VDWCutoff)
        IF(Overlap) WRITE(*,*) "After the simulation is done,Inter-Particle Spatial
Overlap observed."

 ! Calculation of a Vertical Profile of Volume Fraction
 !
```

```
        ConcAvg = DBLE(Hist)/(DBLE(Nstep*(NDdataCollect-
1)/NDdataCollect)/DBLE(Nupdate))&
       *(4.0D0/3.0D0)*Pi/(BoxSize(1)*BoxSize(2)*Gap)
         PSI_AVG = PSI_AVG / DBLE(Nstep*(NDdataCollect-1)/NDdataCollect)

        SResistance = 0
        DO ibin = -Nbin, Nbin
                SResistance  = SResistance  + ConcAvg(ibin) * Happel(ConcAvg(ibin))
        END Do
        SResistance = 1.0D0 / DBLE(2*Nbin+1) * SResistance * 9.0D0 / (2.0D0 *
RadiusP**2.0D0)

!=====================================================================
!=====================================================================
!=====================================================================
!=====================================================================
!=====================================================================

        OPEN(unit = 80,file=ExtraData,status="REPLACE")
        DO ip=1,NPmax
                WRITE(80,*) SteadyColloid(ip,1), SteadyColloid(ip,2),
SteadyColloid(ip,3), Colloid(ip)%Coord(1), Colloid(ip)%Coord(2),
Colloid(ip)%Coord(3),  MaxMoveSize(ip,1), MaxMoveSize(ip,2), MaxMoveSize(ip,3)
        END DO
        CLOSE(unit = 80)

!== This file contains output part of the code.
        INCLUDE "HSout.f90"
END IF
CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
CALL MPI_TYPE_FREE(ParticleType,ierr)
CALL MPI_FINALIZE(ierr)
!=====================================================================
!=====================================================================
!=====================================================================
!=====================================================================
!=====================================================================

!9999 CONTINUE

!STOP
END program HSCFMC


!=============================================================
```

HS15sub.f90

```
!=====================================================
DOUBLE PRECISION FUNCTION XMIN (X,Y)
!=====================================================
 DOUBLE PRECISION X, Y
 XMIN = MIN(X,Y)

END FUNCTION XMIN


!=====================================================
SUBROUTINE CalculateFDiffZ0 (Nbin, Conc, Gap, FDiffZ0)
!=====================================================
 IMPLICIT NONE
 INTEGER :: ibin, Nbin
 DOUBLE PRECISION    :: Gap, ConcDiff, ConcMean
 DOUBLE PRECISION, DIMENSION(-Nbin:Nbin) :: Conc, FDiffZ0

 DO ibin = -Nbin, Nbin
  IF(Conc(ibin) > 0.0) THEN
   IF(ibin > 0) THEN
    ConcDiff     =   Conc(ibin)   - Conc(ibin-1)
    ConcMean     = ( Conc(ibin)   + Conc(ibin-1) ) / 2.0D0
    FDiffZ0(ibin) =   ConcDiff / Gap / ConcMean
!    FDiffZ0(ibin) =  ( Conc(ibin)   - Conc(ibin-1) ) /   Gap  &
!                / ( Conc(ibin)   + Conc(ibin-1) )

   ELSEIF(ibin == 0) THEN
!    FDiffZ0(ibin) =  ( Conc(ibin+1) - Conc(ibin-1) ) / ( Gap * 2.0D0 ) / Conc(ibin)
    ConcDiff     =   Conc(ibin+1)   - Conc(ibin-1)
    ConcMean     = ( Conc(ibin+1)   + Conc(ibin-1) ) / 2.0D0
    FDiffZ0(ibin) =   ConcDiff / (2.0D0*Gap) / ConcMean


   ELSEIF(ibin < 0) THEN
    ConcDiff     =   Conc(ibin+1) - Conc(ibin)
    ConcMean     = ( Conc(ibin+1) + Conc(ibin) ) / 2.0D0
    FDiffZ0(ibin) =   ConcDiff / Gap / ConcMean
!    FDiffZ0(ibin) =  ( Conc(ibin+1) - Conc(ibin)   ) /   Gap        / Conc(ibin)


   END IF
  ELSE
   FDiffZ0(ibin) = 0.0D0
  END IF
```

```
  END DO
!
END SUBROUTINE CalculateFDiffZ0

!============================================================
SUBROUTINE CalculateConc (NP,Sphere,BoxSize,Nbin,Gap,Conc,HistConc,Rank)
!============================================================
  IMPLICIT NONE
  INCLUDE "HSTypes.h"
  INCLUDE "HSConst.h"

  TYPE(particle), DIMENSION(*)        :: Sphere
  INTEGER                             :: NP
  INTEGER                             :: Nbin
  INTEGER                                    :: Rank
  DOUBLE PRECISION                                 :: Gap
  DOUBLE PRECISION, DIMENSION(3)                   :: BoxSize
!*******
  INTEGER,       DIMENSION(-Nbin:Nbin) :: HistConc
  DOUBLE PRECISION,          DIMENSION(-Nbin:Nbin) :: Conc
  INTEGER                             :: jp, ibin

  HistConc = 0
  DO jp = 1, NP
    ibin = INT(  (Sphere(jp)%Coord(3) + 0.5D0*BoxSize(3)) / Gap ) - Nbin
    HistConc(ibin) = HistConc(ibin) + 1
  END DO

  Conc = DBLE(HistConc)*(4.0D0/3.0D0)*Pi/(BoxSize(1)*BoxSize(2)*Gap)

END SUBROUTINE CalculateConc

!============================================================
SUBROUTINE CalculateConcOld (NP,Sphere,BoxSize,Nbin,Gap,Conc)
!============================================================
  IMPLICIT NONE
  INCLUDE "HSTypes.h"
  INCLUDE "HSConst.h"

  TYPE(particle), DIMENSION(*)        :: Sphere
  INTEGER                             :: NP
  INTEGER                             :: Nbin
  DOUBLE PRECISION                                 :: Gap
  DOUBLE PRECISION, DIMENSION(3)                   :: BoxSize
!*******
  INTEGER,       DIMENSION(-Nbin:Nbin) :: HistConc
```

```
 DOUBLE PRECISION,        DIMENSION(-Nbin:Nbin) :: Conc
 INTEGER                  :: jp, ibin

 HistConc = 0
 DO jp = 1, NP
  ibin = INT(  (Sphere(jp)%Coord(3) + 0.5D0*BoxSize(3)) / Gap ) - Nbin
  HistConc(ibin) = HistConc(ibin) + 1
 END DO

 Conc = DBLE(HistConc)*(4.0D0/3.0D0)*Pi/(BoxSize(1)*BoxSize(2)*Gap)

END SUBROUTINE CalculateConcOld



!=============================================================================
===
DOUBLE PRECISION FUNCTION Happel(phi)
!=============================================================================
===
! This function Happel calculates the Happel correction factor (>1)
! as a function of local volume fraction
!
 IMPLICIT NONE
 DOUBLE PRECISION phi

 Happel = 1.0D0 + 2.0D0/3.0D0*phi**(5.0D0/3.0D0)
 Happel = Happel / (1.0D0 - 1.5D0*phi**(1.0D0/3.0D0)+1.5D0*phi**(5.0D0/3.0D0)-
phi*phi)

END FUNCTION Happel

!=============================================================================
===
DOUBLE PRECISION FUNCTION OPgradientCnhStr(phi)
!=============================================================================
===
 IMPLICIT NONE
! This function calculates osmotic pressure gradient of hard spheres
! with given volume fraction using Carnahan-Starling equation.

 DOUBLE PRECISION phi

 OPgradientCnhStr = (1.0D0 + 4.0D0*phi + 4.0D0*phi**2.0D0 - 4.0D0*phi**3.0D0 +
phi**4.0D0 )
 OPgradientCnhStr = OPgradientCnhStr / ( 1.0D0 - phi)**4.0D0
```

```
END FUNCTION OPgradientCnhStr

!=====================================================================
===
DOUBLE PRECISION FUNCTION SIcorrection (phi)
!=====================================================================
===
  IMPLICIT NONE
  DOUBLE PRECISION phi

  SIcorrection = (1.0D0/3.0D0) * phi**2.0D0 * (1.0D0 + 0.5D0*EXP(8.8D0*phi))

END FUNCTION SIcorrection

!=====================================================================
===
DOUBLE PRECISION FUNCTION ViscoCorrection (phi)
!=====================================================================
===
  IMPLICIT NONE
  DOUBLE PRECISION phi

  IF( phi >= 0.58D0 ) phi = 0.579D0  ! This is to avoid a numberial divergence.

  ViscoCorrection = (  1.0D0 + 1.5D0 * phi / ( 1.0D0 - phi / 0.58D0)  )**2

END FUNCTION ViscoCorrection


!=====================================================================
===
SUBROUTINE  SetupMovie
(BoxSize,Nbin,Gap,MyJobTime,iSeqFileIni,iSeqFileFnl,Grid,InitStructure)
!=====================================================================
===
  IMPLICIT NONE

  DOUBLE PRECISION, DIMENSION(3) :: BoxSize
  INTEGER          :: iFileMovie = 999, iFileBox = 998, iFileVMD = 997
  INTEGER          :: iSeqFileIni, iSeqFileFnl
  INTEGER          :: Nbin, ibin
  DOUBLE PRECISION            :: Gap
  CHARACTER(1)      :: Grid
  CHARACTER(19)     :: MyJobTime
  CHARACTER(60)     :: MovieScript, BoxScript, VMDcommand, VMDshort
```

```
CHARACTER (LEN= 1) :: InitStructure

IF(InitStructure=="R") THEN
 WRITE(MovieScript,*) "MC_"//TRIM(MyJobTime)//"_HS_VMD_movie.tcl"
 WRITE(BoxScript,*)   "MC_"//TRIM(MyJobTime)//"_HS_VMD_cubic.tcl"
 WRITE(VMDcommand,*)  "MC_"//TRIM(MyJobTime)//"_HS_VMD_source.tcl"

ELSE IF(InitStructure=="P") THEN
 WRITE(MovieScript,*) "MC_"//TRIM(MyJobTime)//"_HS_VMD_movie.tcl"
 WRITE(BoxScript,*)   "MC_"//TRIM(MyJobTime)//"_HS_VMD_cubic.tcl"
 WRITE(VMDcommand,*)  "MC_"//TRIM(MyJobTime)//"_HS_VMD_source.tcl"

END IF

OPEN(iFileMovie,file=MovieScript)
  WRITE(iFileMovie,*)'proc movie {start end fileformat} {'
  WRITE(iFileMovie,*)'  set filename [format $fileformat [expr $start]]'
  WRITE(iFileMovie,*)'  incr start'
  WRITE(iFileMovie,*)'  puts "Reading initial frame in xyz sequence $filename"'
  WRITE(iFileMovie,*)'  mol load xyz $filename'
  WRITE(iFileMovie,*)"
  WRITE(iFileMovie,*)'  puts "Reading xyz files as an animation..."'
  WRITE(iFileMovie,*)'  for {set i $start} {$i <= $end} {incr i 1} {'
  WRITE(iFileMovie,*)'    set filename [format $fileformat [expr $i]]'
  WRITE(iFileMovie,*)'    animate read xyz $filename'
  WRITE(iFileMovie,*)'  }'
  WRITE(iFileMovie,*)'}'
  WRITE(iFileMovie,*)"
CLOSE(iFileMovie)

OPEN(iFileBox,file=BoxScript)
  WRITE(iFileBox,*)'proc box [4] {'
  WRITE(iFileBox,*)'     set minx ', - BoxSize(1)/2.0D0
  WRITE(iFileBox,*)'     set maxx ',   BoxSize(1)/2.0D0
  WRITE(iFileBox,*)'     set miny ', - BoxSize(2)/2.0D0
  WRITE(iFileBox,*)'     set maxy ',   BoxSize(2)/2.0D0
  WRITE(iFileBox,*)'     set minz ', - BoxSize(3)/2.0D0
  WRITE(iFileBox,*)'     set maxz ',   BoxSize(3)/2.0D0
  WRITE(iFileBox,*)"
  WRITE(iFileBox,*)'     # and draw the lines'
  WRITE(iFileBox,*)'     draw materials off'
  WRITE(iFileBox,*)"
  WRITE(iFileBox,*)'     draw color green '
  WRITE(iFileBox,*)'     draw line "$minx $miny $minz" "$minx $miny $maxz" '
  WRITE(iFileBox,*)'     draw line "$maxx $miny $minz" "$maxx $miny $maxz" '
  WRITE(iFileBox,*)'     draw line "$minx $maxy $minz" "$minx $maxy $maxz" '
```

```
   WRITE(iFileBox,*)'     draw line "$maxx $maxy $maxz" "$maxx $maxy $minz" '
   WRITE(iFileBox,*)' '
   WRITE(iFileBox,*)'     draw color red '
   WRITE(iFileBox,*)'     draw line "$minx $miny $minz" "$maxx $miny $minz" '
   WRITE(iFileBox,*)'     draw line "$minx $miny $minz" "$minx $maxy $minz" '
   WRITE(iFileBox,*)'     draw line "$maxx $miny $minz" "$maxx $maxy $minz" '
   WRITE(iFileBox,*)'     draw line "$minx $maxy $minz" "$maxx $maxy $minz" '
   WRITE(iFileBox,*)'     draw line "$maxx $maxy $maxz" "$minx $maxy $maxz" '
   WRITE(iFileBox,*)'     draw line "$maxx $maxy $maxz" "$maxx $miny $maxz" '
   WRITE(iFileBox,*)'     draw line "$minx $miny $maxz" "$maxx $miny $maxz" '
   WRITE(iFileBox,*)'     draw line "$minx $miny $maxz" "$minx $maxy $maxz" '
   WRITE(iFileBox,*)' '


   IF( Grid == "G" .or. Grid == "g" )  THEN
   DO ibin = 1, 2*Nbin

   WRITE(iFileBox,*)'     set midz ',   BoxSize(3)/2.0D0 - Gap*DBLE(ibin)
   WRITE(iFileBox,*)'     draw color white '
   WRITE(iFileBox,*)'     draw line "$minx $miny $midz" "$maxx $miny $midz" '
   WRITE(iFileBox,*)'     draw line "$minx $miny $midz" "$minx $maxy $midz" '
   WRITE(iFileBox,*)'     draw line "$maxx $miny $midz" "$maxx $maxy $midz" '
   WRITE(iFileBox,*)'     draw line "$minx $maxy $midz" "$maxx $maxy $midz" '
   WRITE(iFileBox,*)' '

   ENDDO
   END IF



   WRITE(iFileBox,*)' } '
  CLOSE(iFileBox)


  OPEN(iFileVMD,file=VMDcommand)
    WRITE(MovieScript,*) "";   WRITE(MovieScript,*)
"MC_"//TRIM(MyJobTime)//"_HS_VMD_movie.tcl"
    WRITE(iFileVMD,*)'source ', MovieScript
    WRITE(BoxScript,*)   "";   WRITE(BoxScript,*)
"MC_"//TRIM(MyJobTime)//"_HS_VMD_cubic.tcl"
    WRITE(iFileVMD,*)'source ', BoxScript
    WRITE(iFileVMD,*)'box all'
    WRITE(VMDshort,*)"MC_"//TRIM(MyJobTime)//"_HS_XYZ%05d.xyz"
    WRITE(iFileVMD,'(" movie ",I4,2X,I4,2X,40A)')iSeqFileIni, iSeqFileFnl ,
TRIM(VMDshort)
    WRITE(iFileVMD,*)  'rotate x by -90'
```

```
  WRITE(iFileVMD,*)  'rotate y by -36'
  WRITE(iFileVMD,*)  'puts "    To continuously rotate the box, type and enter:" '
  WRITE(iFileVMD,*)  'puts "    rotate y by 036000 1 "'
  WRITE(iFileVMD,*)  'puts "    To stop rotating the box, type and enter:" '
  WRITE(iFileVMD,*)  'puts "    rotate y by 0" '
 CLOSE(iFileVMD)

END SUBROUTINE  SetupMovie

!=======================================================================
===
SUBROUTINE  SetupMovieOld
(BoxSize,MyJobTime,iSeqFileIni,iSeqFileFnl,InitStructure)
!=======================================================================
===
 IMPLICIT NONE

 DOUBLE PRECISION, DIMENSION(3) :: BoxSize
 INTEGER          :: iFileMovie = 999, iFileBox = 998, iFileVMD = 997
 INTEGER          :: iSeqFileIni, iSeqFileFnl
 CHARACTER(19)     :: MyJobTime
 CHARACTER(60)     :: MovieScript, BoxScript, VMDcommand, VMDshort
 CHARACTER (LEN= 1) :: InitStructure

 IF(InitStructure=="R") THEN
  WRITE(MovieScript,*) "MC_"//TRIM(MyJobTime)//"_HS_VMD_movie.tcl"
  WRITE(BoxScript,*)  "MC_"//TRIM(MyJobTime)//"_HS_VMD_cubic.tcl"
  WRITE(VMDcommand,*)  "MC_"//TRIM(MyJobTime)//"_HS_VMD_source.tcl"

 ELSE IF(InitStructure=="P") THEN
  WRITE(MovieScript,*) "MC_"//TRIM(MyJobTime)//"_HS_VMD_movie.tcl"
  WRITE(BoxScript,*)  "MC_"//TRIM(MyJobTime)//"_HS_VMD_cubic.tcl"
  WRITE(VMDcommand,*)  "MC_"//TRIM(MyJobTime)//"_HS_VMD_source.tcl"
 END IF


 OPEN(iFileMovie,file=MovieScript)
  WRITE(iFileMovie,*)'proc movie {start end fileformat} {'
  WRITE(iFileMovie,*)'  set filename [format $fileformat [expr $start]]'
  WRITE(iFileMovie,*)'  incr start'
  WRITE(iFileMovie,*)'  puts "Reading initial frame in xyz sequence $filename"'
  WRITE(iFileMovie,*)'  mol load xyz $filename'
  WRITE(iFileMovie,*)"
  WRITE(iFileMovie,*)'  puts "Reading xyz files as an animation..."'
  WRITE(iFileMovie,*)'  for {set i $start} {$i <= $end} {incr i 1} {'
  WRITE(iFileMovie,*)'    set filename [format $fileformat [expr $i]]'
```

```
  WRITE(iFileMovie,*)'    animate read xyz $filename'
  WRITE(iFileMovie,*)'  }'
  WRITE(iFileMovie,*)'}'
  WRITE(iFileMovie,*)"
 CLOSE(iFileMovie)

 OPEN(iFileBox,file=BoxScript)
  WRITE(iFileBox,*)'proc box [4] {'
  WRITE(iFileBox,*)'     set minx ', - BoxSize(1)/2.0D0
  WRITE(iFileBox,*)'     set maxx ',   BoxSize(1)/2.0D0
  WRITE(iFileBox,*)'     set miny ', - BoxSize(2)/2.0D0
  WRITE(iFileBox,*)'     set maxy ',   BoxSize(2)/2.0D0
  WRITE(iFileBox,*)'     set minz ', - BoxSize(3)/2.0D0
  WRITE(iFileBox,*)'     set maxz ',   BoxSize(3)/2.0D0
  WRITE(iFileBox,*)"
  WRITE(iFileBox,*)'     # and draw the lines'
  WRITE(iFileBox,*)'     draw materials off'
  WRITE(iFileBox,*)"
  WRITE(iFileBox,*)'     draw color green '
  WRITE(iFileBox,*)'     draw line "$minx $miny $minz" "$minx $miny $maxz" '
  WRITE(iFileBox,*)'     draw line "$maxx $miny $minz" "$maxx $miny $maxz" '
  WRITE(iFileBox,*)'     draw line "$minx $maxy $minz" "$minx $maxy $maxz" '
  WRITE(iFileBox,*)'     draw line "$maxx $maxy $maxz" "$maxx $maxy $minz" '
  WRITE(iFileBox,*)' '
  WRITE(iFileBox,*)'     draw color red '
  WRITE(iFileBox,*)'     draw line "$minx $miny $minz" "$maxx $miny $minz" '
  WRITE(iFileBox,*)'     draw line "$minx $miny $minz" "$minx $maxy $minz" '
  WRITE(iFileBox,*)'     draw line "$maxx $miny $minz" "$maxx $maxy $minz" '
  WRITE(iFileBox,*)'     draw line "$minx $maxy $minz" "$maxx $maxy $minz" '
  WRITE(iFileBox,*)'     draw line "$maxx $maxy $maxz" "$minx $maxy $maxz" '
  WRITE(iFileBox,*)'     draw line "$maxx $maxy $maxz" "$maxx $miny $maxz" '
  WRITE(iFileBox,*)'     draw line "$minx $miny $maxz" "$maxx $miny $maxz" '
  WRITE(iFileBox,*)'     draw line "$minx $miny $maxz" "$minx $maxy $maxz" '
  WRITE(iFileBox,*)' '
  WRITE(iFileBox,*)' } '
 CLOSE(iFileBox)


 OPEN(iFileVMD,file=VMDcommand)
  WRITE(MovieScript,*) """;   WRITE(MovieScript,*)
"MC_"//TRIM(MyJobTime)//"_HS_VMD_movie.tcl"
  WRITE(iFileVMD,*)'source ', MovieScript
  WRITE(BoxScript,*)   """;   WRITE(BoxScript,*)
"MC_"//TRIM(MyJobTime)//"_HS_VMD_cubic.tcl"
  WRITE(iFileVMD,*)'source ', BoxScript
  WRITE(iFileVMD,*)'box all'
```

```
   WRITE(VMDshort,*)"MC_"//TRIM(MyJobTime)//"_HS_XYZ%05d.xyz"
   WRITE(iFileVMD,'(" movie ",I4,2X,I4,2X,40A)')iSeqFileIni, iSeqFileFnl ,
TRIM(VMDshort)
   WRITE(iFileVMD,*)  'rotate x by -90'
   WRITE(iFileVMD,*)  'rotate y by -36'
   WRITE(iFileVMD,*)  'puts "    To continuously rotate the box, type and enter:" '
   WRITE(iFileVMD,*)  'puts "    rotate y by 036000 1 "'
   WRITE(iFileVMD,*)  'puts "    To stop rotating the box, type and enter:" '
   WRITE(iFileVMD,*)  'puts "    rotate y by 0" '
  CLOSE(iFileVMD)

END SUBROUTINE  SetupMovieOld



!=============================================================
===
SUBROUTINE  PrintColloidCoord(iSeqFile,MyJobTime,NPini,NP,Sphere,InitStructure)
!=============================================================
===
  IMPLICIT NONE
  INCLUDE "HSTypes.h"
  TYPE(particle), DIMENSION(*) :: Sphere
  INTEGER :: ip, NP, NPini
  CHARACTER(19)       :: MyJobTime ! Do not change the number, 19
  CHARACTER(60)       :: SeqFileString
  INTEGER :: iFile, iSeqFile, iSeqFileMax = 100000
  INTEGER :: r, s
!  INTEGER :: scan, len
  CHARACTER (LEN= 1) :: InitStructure
!  INTEGER :: scan, len, mod

  iFile = iSeqFileMax + 1
  WRITE(SeqFileString,'(F6.5)') DBLE(iSeqFile)/DBLE(iSeqFileMax)
  r=scan(SeqFileString,".") ;  s=len(SeqFileString)  ;  SeqFileString=SeqFileString(r+1:s)

  IF(InitStructure=="R") THEN

    WRITE(SeqFileString,*) &
   "MC_"//TRIM(MyJobTime)//"_HS_XYZ"//TRIM(SeqFileString)//".xyz"

  ELSE IF(InitStructure=="P") THEN
    WRITE(SeqFileString,*) &

"MC_"//TRIM(MyJobTime)//"_HS_XYZ"//TRIM(SeqFileString)//".xyz"
```

```
 END IF

 OPEN(iFile,file=SeqFileString)
  WRITE(iFile,*) NP ; WRITE(iFile,*) "!"

!   DO ip = 1, NP
!    IF(mod(ip,10) == 0) THEN
!      WRITE(iFile,*) "A", Sphere(ip)%Coord
!    ELSE
!      WRITE(iFile,*) "B", Sphere(ip)%Coord
!    END IF
!   END DO

   DO ip = 1, NP
    IF(ip <= NPini) THEN
     WRITE(iFile,'(" B ",3(2X,E16.8))') Sphere(ip)%Coord
    ELSE
     WRITE(iFile,'(" A ",3(2X,E16.8))') Sphere(ip)%Coord
    END IF
   END DO
!WRITE(iFileVMD,'(" movie ",I4,2X,I4,2X,40A)')
 CLOSE(iFile)

 iSeqFile= iSeqFile + 1

END SUBROUTINE PrintColloidCoord


!=====================================================================
===
SUBROUTINE UpdateCrossFlowBias (xcatma, xcm, xratio, CrossFlowBias)
!=====================================================================
===
 IMPLICIT NONE
 DOUBLE PRECISION :: xcatma, xcm, xratio, CrossFlowBias

 xratio = xcatma / xcm
 IF(xratio > 0.5D0 ) THEN
  CrossFlowBias =  CrossFlowBias * 1.05D0
 ELSE
  CrossFlowBias =  CrossFlowBias * 0.95D0
 END IF

 xcm    = 0.0D0
 xcatma = 0.0D0
```

```
END SUBROUTINE UpdateCrossFlowBias


!=================================================================
=
SUBROUTINE UpdateDisplacement (acatma, acm, aratio, Displacement)
!=================================================================
=
 IMPLICIT NONE
 DOUBLE PRECISION :: acatma, acm, aratio
 DOUBLE PRECISION :: Displacement

 aratio = acatma / acm
 IF(aratio > 0.5D0 ) THEN
  Displacement =  Displacement * 1.05D0
 ELSE
  Displacement =  Displacement * 0.95D0
 END IF

 acm    = 0.0D0
 acatma = 0.0D0

END SUBROUTINE UpdateDisplacement



!=========================================================
SUBROUTINE UpdateHistogram (NP,Sphere,BoxSize,Gap,Hist,Nbin)
!=========================================================
 IMPLICIT NONE
 INCLUDE "HSTypes.h"
 TYPE(particle), DIMENSION(*)         :: Sphere
 INTEGER                              :: Nbin
 DOUBLE PRECISION                                :: Gap
 INTEGER,       DIMENSION(-Nbin:Nbin) :: Hist
 INTEGER                              :: NP
 INTEGER                              :: jp, ibin
 DOUBLE PRECISION, DIMENSION(3)                  :: BoxSize

 DO jp = 1, NP
  ibin = INT(  (Sphere(jp)%Coord(3) + 0.5D0*BoxSize(3)) / Gap ) - Nbin


!   ibin = INT(  Sphere(jp)%Coord(3)  / Gap )
  Hist(ibin) = Hist(ibin) + 1
 END DO
```

```
END SUBROUTINE UpdateHistogram


!=======================================================================
========
SUBROUTINE CheckOverlapInChannel (Sphere ,NP ,BoxSize ,Overlap, Message,
VDWCutoff)
!=======================================================================
========
!
! This subroutine is to check particle Overlap
! in the membrane channel (slit) implementing
! the periodic boundary conditions in x- and y-directions and
! the impermeable boundary conditions in z-dirction
! through subroutine "CheckOverlapI (i,Sphere,NP,BoxSize,Overlap,Message)".
!
 IMPLICIT NONE
 INCLUDE "HSTypes.h"
 TYPE(particle), DIMENSION(*) :: Sphere
 DOUBLE PRECISION,         DIMENSION(3) :: BoxSize
 DOUBLE PRECISION         :: VDWCutoff
 LOGICAL :: Overlap, Message
 INTEGER :: NP
 INTEGER :: i

 Overlap = .FALSE.
!
 DO i = 1, NP-1
  CALL CheckOverlapI (i,Sphere,NP,BoxSize,Overlap, Message,VDWCutoff)
  IF(Overlap) EXIT
 END DO
!

END SUBROUTINE CheckOverlapInChannel


!=======================================================================
SUBROUTINE CheckOverlapI (i,Sphere,NP,BoxSize,Overlap,Message,VDWCutoff)
!=======================================================================
!
! This subroutine checks overlap between particle i and j = 1+1, NP
! in the membrane channel (slit) implementing
! the periodic boundary conditions in x- and y-directions and
! the impermeable boundary conditions in z-dirction.
!
```

```
 IMPLICIT NONE
 INCLUDE "HSTypes.h"
 TYPE(particle), DIMENSION(*) :: Sphere
 DOUBLE PRECISION,          DIMENSION(*) :: BoxSize
 INTEGER :: i, NP
 LOGICAL :: Overlap, Message
!
 DOUBLE PRECISION         :: VDWCutoff

 INTEGER :: j
 LOGICAL :: OverlapTest
 DOUBLE PRECISION, DIMENSION(3) :: DistIJ
 TYPE(particle)     :: SphereIJ, SphereZero
!  DOUBLE PRECISION              :: dot_product
!
 SphereZero%Coord = 0.0D0
 SphereZero%Radius= 0.0D0
 Overlap = .FALSE.
!
 DO j =  i+1, NP
!    Distance between two particle (i and j) in each direction
    DistIJ      = Sphere(i)%Coord - Sphere(j)%Coord
!
!    Periodic boundary conditions are used only in x- and y- directions
!    to check particle overlap.
    DistIJ(1)    = DistIJ(1) - ANINT(DistIJ(1)/BoxSize(1))*BoxSize(1)
    DistIJ(2)    = DistIJ(2) - ANINT(DistIJ(2)/BoxSize(2))*BoxSize(2)
!
!     DistIJ(3)    = DistIJ(3) - ANINT(DistIJ(3)/BoxSize(3))*BoxSize(3)
!    (This line is commented out NOT to implement the periodic boundary
!     condition in z-direction, but impermeable boundary condition)
!
    SphereIJ%Coord = DistIJ
    SphereIJ%Radius= Sphere(i)%Radius + Sphere(j)%Radius
!
    CALL CheckOverlapIJ (SphereIJ,SphereZero,OverlapTest, VDWCutoff)
!


    IF(OverlapTest==.TRUE.) THEN
       Overlap = .TRUE.
       IF(Message) WRITE(*,*)"Overlap between :", i,j,dot_product(DistIJ,DistIJ)
       EXIT
    ELSE
       Overlap = .FALSE.
    END IF
 END DO
```

```
END SUBROUTINE CheckOverlapI


!=============================================================
SUBROUTINE CheckOverlapIJ (SphereI,SphereJ,OverlapTest,VDWCutoff)
!=============================================================
!
! This subroutine checks whether the center-to-center
! distance between two particles i, and j is less than
! the summation of their radii.
! The periodic boundary condition is not implemented here.
!
 IMPLICIT NONE
 INCLUDE "HSTypes.h"
 TYPE(particle)    :: SphereI, SphereJ
 LOGICAL           :: OverlapTest
!  DOUBLE PRECISION             :: dot_product
 DOUBLE PRECISION, DIMENSION(3) :: DistIJ
 DOUBLE PRECISION             :: SqrDistIJ, SqrDistMinIJ
 DOUBLE PRECISION                    :: VDWCutoff

 DistIJ      = SphereI%Coord - SphereJ%Coord
 SqrDistIJ   = SQRT(dot_product(DistIJ,DistIJ))
 SqrDistMinIJ = SphereI%Radius + SphereJ%Radius + VDWCutoff

 IF(SqrDistIJ <= SqrDistMinIJ) THEN
  OverlapTest = .TRUE.
 ELSE
  OverlapTest = .FALSE.
 END IF

END SUBROUTINE  CheckOverlapIJ

!=============================================================
SUBROUTINE CheckOverlapIJPer (SphereI,SphereJ,BoxSize,OverlapTest,VDWCutoff)
!=============================================================
!
! This subroutine checks whether the center-to-center
! distance between two particles i, and j is less than
! the summation of their radii.
! The periodic boundary condition is implemented here.
!
 IMPLICIT NONE
 INCLUDE "HSTypes.h"
 TYPE(particle)    :: SphereI, SphereJ
```

```
  LOGICAL          :: OverlapTest
!  DOUBLE PRECISION            :: dot_product
  DOUBLE PRECISION,         DIMENSION(*) :: BoxSize
  DOUBLE PRECISION, DIMENSION(3) :: DistIJ
  DOUBLE PRECISION            :: SqrDistIJ, SqrDistMinIJ
  DOUBLE PRECISION                  :: VDWCutoff

  DistIJ      = SphereI%Coord - SphereJ%Coord
!    Periodic boundary conditions are used only in x- and y- directions
!    to check particle overlap.
  DistIJ(1)    = DistIJ(1) - ANINT(DistIJ(1)/BoxSize(1))*BoxSize(1)
  DistIJ(2)    = DistIJ(2) - ANINT(DistIJ(2)/BoxSize(2))*BoxSize(2)
  SqrDistIJ    = SQRT(dot_product(DistIJ,DistIJ))
  SqrDistMinIJ = SphereI%Radius + SphereJ%Radius + VDWCutoff

  IF(SqrDistIJ <= SqrDistMinIJ) THEN
    OverlapTest = .TRUE.
  ELSE
    OverlapTest = .FALSE.
  END IF

END SUBROUTINE  CheckOverlapIJPer

!=======================================
SUBROUTINE InitCoordRect (Sphere,NP,BoxSize)
!=======================================
  IMPLICIT NONE

  INCLUDE "HSTypes.h"
  DOUBLE PRECISION,         DIMENSION(3) :: BoxSize
  TYPE(particle), DIMENSION(*) :: Sphere
  INTEGER :: NP

  INTEGER :: Nx, Ny, Nz, Nxyz
  INTEGER :: i, j, k, n

!  INTEGER :: NPcubic1, NPcubic3
  DOUBLE PRECISION    :: DCC = 2.1D0

  DOUBLE PRECISION, DIMENSION (3)    :: CM

  Ny = INT( ( DBLE(NP) *  BoxSize(2)**2.0D0 / (BoxSize(1) * BoxSize(3))
)**(1.0D0/3.0D0) + 0.5D0 )
  Nx = INT(   DBLE(Ny) *  BoxSize(1)    /  BoxSize(2) + 0.5D0 )
  Nz = INT(   DBLE(Ny) *  BoxSize(3)    /  BoxSize(2) + 0.5D0 )
  Nxyz = Nx*Ny* Nz
```

```
! write(*,*) BoxSize
! write(*,*) NP, Nx, Ny, Nz,Nxyz


 CM = 0.0
 DO k = 1, Nz
  DO j = 1, Ny
   DO i = 1, Nx
     n = i + (j-1)*Nx + (k-1)*Nx*Ny
     IF ( n <= NP) THEN
       Sphere(n)%Coord(1) = DBLE(i)*DCC
       Sphere(n)%Coord(2) = DBLE(j)*DCC
       Sphere(n)%Coord(3) = DBLE(k)*DCC
       CM(3) = CM(3) + Sphere(n)%Coord(3)
       CM(2) = CM(2) + Sphere(n)%Coord(2)
       CM(1) = CM(1) + Sphere(n)%Coord(1)
     END IF
   END DO
  END DO
 END DO

 DO k = 1, 3
  CM(k) = CM(k) / DBLE(NP)
 END DO

 DO k = 1, Nz
  DO j = 1, Ny
   DO i = 1, Nx
     n = i + (j-1)*Nx + (k-1)*Nx*Ny
     IF ( n <= NP) THEN
       Sphere(n)%Coord(1) =  Sphere(n)%Coord(1) - CM(1)
       Sphere(n)%Coord(2) =  Sphere(n)%Coord(2) - CM(2)
       Sphere(n)%Coord(3) =  Sphere(n)%Coord(3) - CM(3)
     END IF
   END DO
  END DO
 END DO

END SUBROUTINE InitCoordRect

!=======================================
SUBROUTINE InitCoordCubic (Sphere,NP)
!=======================================
 IMPLICIT NONE
```

```
 INCLUDE "HSTypes.h"
 TYPE(particle), DIMENSION(*) :: Sphere
 INTEGER :: NP

 INTEGER :: NPcubic1, NPcubic3, k
 DOUBLE PRECISION    :: DCC = 2.1D0

 DOUBLE PRECISION, DIMENSION (3)    :: CM

 CM = 0.0
 NPcubic1 = INT((DBLE(NP))**(1.0D0/3.0D0) + 0.5D0 )
 NPcubic3 = NPcubic1**3.0D0
 DO k = 1, NP
    Sphere(k)%Coord(3) = DCC*DBLE( MOD((k-1)/NPcubic1**0,NPcubic1) )
    Sphere(k)%Coord(2) = DCC*DBLE( MOD((k-1)/NPcubic1**1,NPcubic1) )
    Sphere(k)%Coord(1) = DCC*DBLE( MOD((k-1)/NPcubic1**2,NPcubic1) )
    CM(3) = CM(3) + Sphere(k)%Coord(3)
    CM(2) = CM(2) + Sphere(k)%Coord(2)
    CM(1) = CM(1) + Sphere(k)%Coord(1)
 END DO

 DO k = 1, 3
   CM(k) = CM(k) / DBLE(NP)
 END DO

 DO k = 1, NP
    Sphere(k)%Coord(3) =  Sphere(k)%Coord(3) - CM(3)
    Sphere(k)%Coord(2) =  Sphere(k)%Coord(2) - CM(2)
    Sphere(k)%Coord(1) =  Sphere(k)%Coord(1) - CM(1)
 END DO

END SUBROUTINE InitCoordCubic



!=======================================
SUBROUTINE InitCoordCubicOld (Sphere,NP)
!=======================================
 IMPLICIT NONE

 INCLUDE "HSTypes.h"
 TYPE(particle), DIMENSION(*) :: Sphere
 INTEGER :: NP

 INTEGER :: NPcubic1, NPcubic3, k
 DOUBLE PRECISION    :: DCC = 2.1D0
```

```
 NPcubic1 = INT((DBLE(NP))**(1.0D0/3.0D0) + 1.0D0 )
 NPcubic3 = NPcubic1**3.0D0
 DO k = 1, NP
    Sphere(k)%Coord(3) = DCC*DBLE( MOD((k-1)/NPcubic1**0,NPcubic1) ) -
DCC*DBLE(NPcubic1-1)/2.0D0
    Sphere(k)%Coord(2) = DCC*DBLE( MOD((k-1)/NPcubic1**1,NPcubic1) ) -
DCC*DBLE(NPcubic1-1)/2.0D0
    Sphere(k)%Coord(1) = DCC*DBLE( MOD((k-1)/NPcubic1**2,NPcubic1) ) -
DCC*DBLE(NPcubic1-1)/2.0D0
 END DO


END SUBROUTINE InitCoordCubicOld


!=======================================
SUBROUTINE InitCoordRandom (Sphere,NP,Surf,SurfPart,BoxSize,VDWCutoff)
!=======================================
!
! This sburoutine is to initially distribute
! NP Particles in the simulation Box
! in a Random manner.
!
! Centers of all the particles are located
! in a way that any surfaces of spheres do not
! touch or trespass the box boundaries.
!
 IMPLICIT NONE
 INCLUDE "HSTypes.h"
 TYPE(particle), DIMENSION(*) :: Sphere
 TYPE(particle), DIMENSION(*) :: SurfPart
 TYPE(particle)               :: SphereA

!
! Coordinates of Sphere(s) are initialized
! using their radii values and BoxSizes
! But, Spheres' Zeta potentail has been
! NEITHER used NOR modified.
!
 DOUBLE PRECISION         :: VDWCutoff
 INTEGER                  :: Surf
 DOUBLE PRECISION,        DIMENSION(*) :: BoxSize
 INTEGER :: NP
 DOUBLE PRECISION    :: RandNum
 INTEGER :: i, j
```

```
  LOGICAL :: Overlap, OverlapTest
!
! For the first particle (index = 1)
!
! To put particles uniformly in x,y,z-directions
!  CALL RANDOM_NUMBER(RandNum)
!  Sphere(1)%Coord(1) = 2.0D0*( 0.5D0 - RandNum ) * ( 0.5D0*BoxSize(1) -
Sphere(1)%Radius )
!  CALL RANDOM_NUMBER(RandNum)
!  Sphere(1)%Coord(2) = 2.0D0*( 0.5D0 - RandNum ) * ( 0.5D0*BoxSize(2) -
Sphere(1)%Radius )
!   CALL RANDOM_NUMBER(RandNum)
!  Sphere(1)%Coord(3) = 2.0D0*( 0.5D0 - RandNum ) * ( 0.5D0*BoxSize(3) -
Sphere(1)%Radius )
!
! For the rest of the particles (index>1)
!
  SphereLoop:&
  DO i = 1, NP

    Overlap = .TRUE.
    DO WHILE (Overlap)
      CALL RANDOM_NUMBER(RandNum)
      Sphere(i)%Coord(1) = 2.0D0*( 0.5D0-RandNum) * (0.5D0*BoxSize(1)-
Sphere(i)%Radius)
      CALL RANDOM_NUMBER(RandNum)
      Sphere(i)%Coord(2) = 2.0D0*( 0.5D0-RandNum) * (0.5D0*BoxSize(2)-
Sphere(i)%Radius)
      CALL RANDOM_NUMBER(RandNum)
      Sphere(i)%Coord(3) = 2.0D0*( 0.5D0-RandNum) * (0.5D0*BoxSize(3)-
Sphere(i)%Radius)

      OverlapTest = .FALSE.
      IF (i.GT.1) THEN
          DO j = 1, i-1
          CALL CheckOverlapIJ (Sphere(i),Sphere(j),OverlapTest,VDWCutoff)
          IF(OverlapTest==.TRUE.) EXIT
          END DO
      ENDIF
      IF (Surf.GT.0) THEN
          DO j = 1, Surf
          IF(OverlapTest==.TRUE.) EXIT
                  CALL
CheckOverlapIJPer(Sphere(i),SurfPart(j),BoxSize,OverlapTest,VDWCutoff)
          END DO
      ENDIF
```

```
    Overlap = OverlapTest

   END DO

  END DO &
  SphereLoop


! Sorting here
   DO i = 1, NP-1
     DO j = i+1, NP

       IF(ABS(Sphere(i)%Coord(3)) < ABS(Sphere(j)%Coord(3))) THEN
           SphereA       = Sphere(i)
           Sphere(i)     = Sphere(j)
           Sphere(j)     = SphereA
       END IF

     END DO
   END DO

END SUBROUTINE InitCoordRandom

!=======================================
SUBROUTINE InsertCoordRandom (SphereFed,NPfed,SlabSize,VDWCutoff)
!=======================================
!
! This sburoutine is to initially distribute
! NP Particles in the simulation Box
! in a Random manner.
!
! Centers of all the particles are located
! in a way that any surfaces of spheres do not
! touch or trespass the box boundaries.
!
  IMPLICIT NONE
  INCLUDE "HSTypes.h"
  TYPE(particle), DIMENSION(*) :: SphereFed
!
! Coordinates of Sphere(s) are initialized
! using their radii values and BoxSizes
! But, Spheres' Zeta potentail has been
! NEITHER used NOR modified.
!
  DOUBLE PRECISION          :: VDWCutoff
```

```
 DOUBLE PRECISION,          DIMENSION(*) :: SlabSize
 INTEGER :: NPfed
 DOUBLE PRECISION    :: RandNum
 INTEGER :: i, j
 LOGICAL :: Overlap, OverlapTest
!
! For the first particle (index = 1)
!
! To put particles uniformly in x,y,z-directions
 CALL RANDOM_NUMBER(RandNum)
 SphereFed(1)%Coord(1) = 2.0D0*( 0.5D0 - RandNum ) * ( 0.5D0*SlabSize(1) -
SphereFed(1)%Radius )
 CALL RANDOM_NUMBER(RandNum)
 SphereFed(1)%Coord(2) = 2.0D0*( 0.5D0 - RandNum ) * ( 0.5D0*SlabSize(2) -
SphereFed(1)%Radius )
 CALL RANDOM_NUMBER(RandNum)
 SphereFed(1)%Coord(3) = 2.0D0*( 0.5D0 - RandNum ) * ( 0.5D0*SlabSize(3) -
SphereFed(1)%Radius )
!
! For the rest of the particles (index>1)
!
 SphereLoop:&
 DO i = 2, NPfed

  Overlap = .TRUE.
  DO WHILE (Overlap)
   CALL RANDOM_NUMBER(RandNum)
   SphereFed(i)%Coord(1) = 2.0D0*( 0.5D0-RandNum) * (0.5D0*SlabSize(1)-
SphereFed(i)%Radius)
   CALL RANDOM_NUMBER(RandNum)
   SphereFed(i)%Coord(2) = 2.0D0*( 0.5D0-RandNum) * (0.5D0*SlabSize(2)-
SphereFed(i)%Radius)
   CALL RANDOM_NUMBER(RandNum)
   SphereFed(i)%Coord(3) = 2.0D0*( 0.5D0-RandNum) * (0.5D0*SlabSize(3)-
SphereFed(i)%Radius)

   OverlapTest = .FALSE.
   DO j = 1, i-1
    CALL CheckOverlapIJ (SphereFed(i),SphereFed(j),OverlapTest,VDWCutoff)
    IF(OverlapTest==.TRUE.) EXIT
   END DO
   Overlap = OverlapTest

  END DO

 END DO &
```

```
  SphereLoop

END SUBROUTINE InsertCoordRandom


!=====================================
SUBROUTINE InitZetaPot(ZetaPot,NP,zeta)
!=====================================
!
! This subroutine is to assign the Zeta Potential
! values of all the Particles
!
  IMPLICIT NONE
  DOUBLE PRECISION,DIMENSION(*) :: ZetaPot
  DOUBLE PRECISION              :: zeta
  INTEGER          :: NP
  INTEGER          :: i

  DO i = 1, NP
    ZetaPot(i) = zeta
  END DO

END SUBROUTINE InitZetaPot


!================================
SUBROUTINE InitRadius(Radius,NP)
!================================
!
! This subroutine is to assign the Raius
! values of all the Particles
!
  IMPLICIT NONE
  DOUBLE PRECISION,DIMENSION(*) :: Radius
  INTEGER          :: NP
  INTEGER          :: i

  DO i = 1, NP
    Radius(i) = 1.0D0
  END DO

END SUBROUTINE InitRadius

!================================
SUBROUTINE InitTime(Time,Moves,NP)
!================================
```

```
!
! This subroutine is to assign the Raius
! values of all the Particles
!
  IMPLICIT NONE
  DOUBLE PRECISION,DIMENSION(*) :: Time
  INTEGER,DIMENSION(*) :: Moves
  INTEGER          :: NP
  INTEGER          :: i

  DO i = 1, NP
    Time(i) = 0.0D0
    Moves(i) = 0
  END DO

END SUBROUTINE InitTime

!=========================================
SUBROUTINE InitBox (BoxSize, NP, VolFrac)
!=========================================
!
! This subroutine is to determine
! Length = BoxSize(1),
! Width  = BoxSize(2), and
! Height = BoxSize(3)
! of the Simulation Box
!
  IMPLICIT NONE
  INCLUDE "HSConst.h"
  DOUBLE PRECISION, DIMENSION(3) :: BoxSize
  DOUBLE PRECISION               :: VolFrac
!  DOUBLE PRECISION               :: float
  INTEGER          :: NP

  BoxSize(1) = ( 4.0D0*Pi*DBLE(NP) / (3.0D0*VolFrac) )**(1.0D0/3.0D0)
  BoxSize(2) = BoxSize(1)
  BoxSize(3) = BoxSize(1)

END SUBROUTINE InitBox

!=========================================
SUBROUTINE InitRecBox (BoxSize, NP, NboxX, NboxZ, VolFrac, FixedBox, FixedH)
!=========================================
  IMPLICIT NONE
  INCLUDE "HSConst.h"
  DOUBLE PRECISION, DIMENSION(3) :: BoxSize
```

```
  DOUBLE PRECISION             :: VolFrac, FixedH
!  DOUBLE PRECISION             :: float
  DOUBLE PRECISION             :: Width
  INTEGER          :: NP, NboxZ, NboxX
  CHARACTER  (LEN= 1) :: FixedBox

IF (FixedBox.EQ."N") THEN
  Width      = ( 4.0D0*Pi*DBLE(NP)/DBLE(NboxZ*NboxX) / (3.0D0*VolFrac)
)**(1.0D0/3.0D0)
  BoxSize (2) =   Width
  BoxSize (1) =   Width * DBLE(NboxX)  !  Length
  BoxSize (3) =   Width * DBLE(NboxZ)  !  Height
ELSEIF (FixedBox.EQ."Y") THEN
  Width      = ( 4.0D0*Pi*DBLE(NP) / (3.0D0*VolFrac) / FixedH / DBLE(NboxX)
)**(1.0D0/2.0D0)
  BoxSize (2) =   Width
  BoxSize (1) =   Width * DBLE(NboxX)  !  Length
  BoxSize (3) =   FixedH  !  Height
ENDIF

END SUBROUTINE InitRecBox

!=====================================================================
SUBROUTINE CheckOverlapI2 (i,Sphere,NP,BoxSize,Overlap,Message,VDWCutoff)
!=====================================================================
!
! This subroutine checks overlap between particle i and j = 1+1, NP
! in the membrane channel (slit) implementing
! the periodic boundary conditions in x- and y-directions and
! the impermeable boundary conditions in z-dirction.
!
  IMPLICIT NONE
  INCLUDE "HSTypes.h"
  TYPE(particle), DIMENSION(*) :: Sphere
  DOUBLE PRECISION,          DIMENSION(*) :: BoxSize
  INTEGER :: i, NP
  LOGICAL :: Overlap, Message
!
  DOUBLE PRECISION          :: VDWCutoff

  INTEGER :: j
  LOGICAL :: OverlapTest
  DOUBLE PRECISION, DIMENSION(3) :: DistIJ
  TYPE(particle)     :: SphereIJ, SphereZero
!  DOUBLE PRECISION             :: dot_product
!
```

```
 SphereZero%Coord = 0.0D0
 SphereZero%Radius= 0.0D0
 Overlap = .FALSE.
!
 DO j =  1, NP
  IF(j.NE.i) THEN
!   Distance between two particle (i and j) in each direction
    DistIJ      = Sphere(i)%Coord - Sphere(j)%Coord
!
!   Periodic boundary conditions are used only in x- and y- directions
!   to check particle overlap.
    DistIJ(1)   = DistIJ(1) - ANINT(DistIJ(1)/BoxSize(1))*BoxSize(1)
    DistIJ(2)   = DistIJ(2) - ANINT(DistIJ(2)/BoxSize(2))*BoxSize(2)
!
!    DistIJ(3)   = DistIJ(3) - ANINT(DistIJ(3)/BoxSize(3))*BoxSize(3)
!   (This line is commented out NOT to implement the periodic boundary
!    condition in z-direction, but impermeable boundary condition)
!
    SphereIJ%Coord = DistIJ
    SphereIJ%Radius= Sphere(i)%Radius + Sphere(j)%Radius
!
    CALL CheckOverlapIJ (SphereIJ,SphereZero,OverlapTest,VDWCutoff)
!

    IF(OverlapTest==.TRUE.) THEN
      Overlap = .TRUE.
      IF(Message) WRITE(*,*)"Overlap between :", i,j,dot_product(DistIJ,DistIJ)
      EXIT
    ELSE
      Overlap = .FALSE.
    END IF

  END IF

 END DO

END SUBROUTINE CheckOverlapI2


!=======================================================================
SUBROUTINE Calculate_Overlap_Energy_Conc_Grad &
       (i,Sphere,NP,BoxSize,Overlap,Message,Energy,LocalConc,VDWCutoff)
!=======================================================================
!
! This subroutine checks overlap between particle i and j = 1+1, NP
! in the membrane channel (slit) implementing
```

```
! the periodic boundary conditions in x- and y-directions and
! the impermeable boundary conditions in z-dirction.
!
 IMPLICIT NONE
 INCLUDE "HSTypes.h"
 INCLUDE "HSConst.h"

 TYPE(particle), DIMENSION(*) :: Sphere
 DOUBLE PRECISION,         DIMENSION(*) :: BoxSize
 DOUBLE PRECISION                       :: LocalBoxLength
 DOUBLE PRECISION                       :: LocalBoxWidth
 DOUBLE PRECISION                       :: LocalBoxHeight
 DOUBLE PRECISION                       :: LocalBoxVolume
 INTEGER :: i, NP
 LOGICAL :: Overlap, Message
!
 INTEGER :: j
 LOGICAL :: OverlapTest
 DOUBLE PRECISION    :: Energy
 DOUBLE PRECISION    :: LocalConc

 DOUBLE PRECISION         :: VDWCutoff

 DOUBLE PRECISION    :: XMIN
 DOUBLE PRECISION, DIMENSION(3) :: DistIJ
 TYPE(particle)     :: SphereIJ, SphereZero
!  DOUBLE PRECISION              :: dot_product
!
 LocalBoxHeight = Sphere(1)%Radius * 8.0D0
 LocalBoxWidth  = Sphere(1)%Radius * 8.0D0
 LocalBoxLength = Sphere(1)%Radius * 8.0D0
!  LocalBoxVolume = BoxSize(1) * BoxSize(2) * LocalBoxHeight
!
 Energy    = 0.0D0
 LocalConc = 1.0D0
!
 SphereZero%Coord = 0.0D0
 SphereZero%Radius= 0.0D0
 Overlap = .FALSE.
!
 DO j =  1, NP
   IFIJ: IF(j.NE.i) THEN

!   [A]  Distance between two particle (i and j) in each direction
     DistIJ       = Sphere(i)%Coord - Sphere(j)%Coord
!
```

```
!   [B] Overlap Test
!    Periodic boundary conditions are used only in x- and y- directions
!    to check particle overlap.
     DistIJ(1)    = DistIJ(1) - ANINT(DistIJ(1)/BoxSize(1))*BoxSize(1)
     DistIJ(2)    = DistIJ(2) - ANINT(DistIJ(2)/BoxSize(2))*BoxSize(2)
!
!     DistIJ(3)    = DistIJ(3) - ANINT(DistIJ(3)/BoxSize(3))*BoxSize(3)
!    (This line is commented out NOT to implement the periodic boundary
!     condition in z-direction, but impermeable boundary condition)
!
     SphereIJ%Coord = DistIJ
     SphereIJ%Radius= Sphere(i)%Radius + Sphere(j)%Radius
!
     CALL CheckOverlapIJ (SphereIJ,SphereZero,OverlapTest,VDWCutoff)
!
     IF(OverlapTest==.TRUE.) THEN
        Overlap = .TRUE.
        IF(Message) WRITE(*,*)"Overlap between :", i,j,dot_product(DistIJ,DistIJ)
        EXIT
     ELSE
        Overlap = .FALSE.
     END IF

!   [C] Engergy Calculation (before applying periodic BC in z-dir)



!   [D] Concentrtion and Concentration Gradient
     IF ( ( DistIJ(1) > - LocalBoxLength/2.0D0 .and. DistIJ(1) <  LocalBoxLength/2.0D0 )
.and. &
          ( DistIJ(2) > - LocalBoxWidth /2.0D0 .and. DistIJ(2) <  LocalBoxWidth /2.0D0 )
.and. &
          ( DistIJ(3) > - LocalBoxHeight/2.0D0 .and. DistIJ(3) <  LocalBoxHeight/2.0D0 ) )
THEN

          LocalConc   = LocalConc   +  1.0D0

     ENDIF

   END IF IFIJ
  END DO

  LocalBoxVolume = LocalBoxLength * LocalBoxWidth &
              * ( LocalBoxHeight/2.0D0 &
                + XMIN(LocalBoxHeight/2.0D0, BoxSize(3)/2.0D0-
ABS(Sphere(i)%Coord(3))))
```

```
  LocalConc = LocalConc * 4.0D0 / 3.0D0 * Pi * (Sphere(1)%Radius)**3.0D0 /
LocalBoxVolume


END SUBROUTINE Calculate_Overlap_Energy_Conc_Grad

!=============================================================
SUBROUTINE VelocityCalc(Nx,Np,Coord,RadiusP,DelP,alpha,eta_0,&
        x,Tx1,Tx2,ConcSlab,ConcCorrection,BoxSize,&
        MaxVelLoc,MaxVel,VelCoeff,ConcCoeff)
!=============================================================
!
!This subroutine is designed to evaluate a 1D velocity profile
!for use in the px_bias for each partical, as well to prepare
!for the solution of the local shear for each particle.
!
IMPLICIT NONE
INCLUDE "HSConst.h"
INCLUDE "/opt/apps/openmpi/1.3.3-intel/include/mpif.h"
INTEGER                                         ::      i, j, Iter, MaxIter
INTEGER                                         ::      Nx, Nx1, Np, Info, InfoC
INTEGER,    DIMENSION(1:Nx+1)                   ::      Ipiv
INTEGER,    DIMENSION(1:4*Nx)                   ::      IpivC
INTEGER                                         ::      ierr
DOUBLE PRECISION                                        ::      RadiusP,
CurrentHeight, HalfHeight, DelP, alpha, eta_0, ConcCorrection
DOUBLE PRECISION                                        ::      OldLoc,
MaxVelLoc, MaxVel, DerVel, Der2Vel, Tol
DOUBLE PRECISION,       DIMENSION(0:Nx)                 ::      x,
ConcSlab
DOUBLE PRECISION,       DIMENSION(0:Nx,0:Nx)            ::      Tx1,
Tx2
DOUBLE PRECISION,       DIMENSION(0:Nx)                 ::      Conc,
Visc
DOUBLE PRECISION,       DIMENSION(1:Nx-1)               ::
        ConcGrad
DOUBLE PRECISION,       DIMENSION(1:Np)                 ::      Coord
DOUBLE PRECISION,       DIMENSION(3)                    ::
        BoxSize
DOUBLE PRECISION,       DIMENSION(1:4*Nx,1:4*Nx)        ::      AC
!DOUBLE PRECISION,      DIMENSION(1:NConc+1,1:NConc+1)          ::
        ACtAC
DOUBLE PRECISION,       DIMENSION(1:4*Nx)               ::      bC,
ConcCoeff
```

```
DOUBLE PRECISION,       DIMENSION(1:Nx+1,1:Nx+1)            ::      A, A1,
A2
DOUBLE PRECISION,       DIMENSION(1:Nx+1)                   ::      b,
VelCoeff
DOUBLE PRECISION,       DIMENSION(0:Nx)                     ::      T, T1,
T2
!DOUBLE PRECISION,      DIMENSION(0:Nx+1)                   ::
        ConcLine
INTEGER                                                     ::      Loc
DOUBLE PRECISION                                            ::      RefHeight,
CapVol, SecVol, CapHeight
EXTERNAL DGESV

HalfHeight=BoxSize(3)/2.0D0
Nx1=Nx !REMNANT OF CAMPO'S CODE
MaxIter=200
A1=0.0D0
A2=0.0D0
A=0.0D0
b=0.0D0
VelCoeff=0.0D0
T=0.0D0
T1=0.0D0
T2=0.0D0
ConcCoeff=0.0D0
AC=0.0D0
bC=0.0D0
!ACtAC=0.0D0
ConcGrad=0.0D0
!CONCENTRATION COMPUTATION
CurrentHeight=x(0)*HalfHeight
!++++++++++++++++++++++++++++++++++++++++++++++++++MODS TO CONC
!ConcLine=0.0D0
Conc=0.0D0
!DO i=0,Nx
!       ConcLine(i)=CurrentHeight
!       CurrentHeight=CurrentHeight-ConcSlab(i)
!ENDDO
!ConcLine(Nx+1)=CurrentHeight
DO i=1,Np
        CurrentHeight=x(0)*HalfHeight
        DO j=0,Nx
                IF((Coord(i).LE.CurrentHeight).AND.(Coord(i).GT.(CurrentHeight-
ConcSlab(j)))) EXIT
                CurrentHeight=CurrentHeight-ConcSlab(j)
        ENDDO
```

```
!ABOVE CENTER
        IF ((Coord(i)+1.0D0).LE.CurrentHeight) THEN
                Conc(j)=Conc(j)+(2.0D0/3.0D0)*Pi
        ELSE
                RefHeight=CurrentHeight
                Loc=j
                CapHeight=1.0D0-(RefHeight-Coord(i))
                CapVol=(1.0D0/3.0D0)*Pi*(CapHeight**2.0D0)*(3.0D0-CapHeight)
                SecVol=(2.0D0/3.0D0)*Pi-CapVol
                Conc(Loc)=Conc(Loc)+SecVol
                DO WHILE (1==1)
                        Loc=Loc-1
                        RefHeight=RefHeight+ConcSlab(Loc)
                        IF ((Coord(i)+1.0D0).LE.RefHeight) THEN
                                Conc(Loc)=Conc(Loc)+CapVol
                                EXIT
                        ELSE
                                CapHeight=1.0D0-(RefHeight-Coord(i))
                                SecVol=CapVol
                                CapVol=(1.0D0/3.0D0)*Pi*(CapHeight**2.0D0)*(3.0D0-
CapHeight)
                                SecVol=SecVol-CapVol
                                Conc(Loc)=Conc(Loc)+SecVol
                        ENDIF
                ENDDO
        ENDIF
!BELOW CENTER
        IF ((Coord(i)-1.0D0).GE.(CurrentHeight-ConcSlab(j))) THEN
                Conc(j)=Conc(j)+(2.0D0/3.0D0)*Pi
        ELSE
                RefHeight=CurrentHeight-ConcSlab(j)
                Loc=j
                CapHeight=1.0D0-(Coord(i)-RefHeight)
                CapVol=(1.0D0/3.0D0)*Pi*(CapHeight**2.0D0)*(3.0D0-CapHeight)
                SecVol=(2.0D0/3.0D0)*Pi-CapVol
                Conc(Loc)=Conc(Loc)+SecVol
                DO WHILE (1==1)
                        Loc=Loc+1
                        RefHeight=RefHeight-ConcSlab(Loc)
                        IF ((Coord(i)-1.0D0).GE.RefHeight) THEN
                                Conc(Loc)=Conc(Loc)+CapVol
                                EXIT
                        ELSE
                                CapHeight=1.0D0-(Coord(i)-RefHeight)
                                SecVol=CapVol
```

```
                                CapVol=(1.0D0/3.0D0)*Pi*(CapHeight**2.0D0)*(3.0D0-
CapHeight)
                                SecVol=SecVol-CapVol
                                Conc(Loc)=Conc(Loc)+SecVol
                        ENDIF
                ENDDO
        ENDIF
ENDDO
!CONC AND VISC CALCS
Conc(0)=0.0D0
DO i = 1,Nx-1
        Conc(i)=Conc(i)/(BoxSize(1)*BoxSize(2)*ConcSlab(i))
        Visc(i)=0.001D0*eta_0*EXP(alpha*Conc(i)*ConcCorrection)
!       Visc(i)=8.56590D-4
END DO
Conc(Nx)=0
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++END MODS TO
CONC
!OLD CONCENTRATION CODE
!DO i=0,Nx-1
!       Conc(i)=0.0D0
!       DO j=1,Np
!               IF((Coord(j).LE.CurrentHeight).AND.(Coord(j).GT.(CurrentHeight-
ConcSlab(i)))) THEN
!                       Conc(i)=Conc(i)+1.0D0
!               END IF
!       END DO
!       CurrentHeight=CurrentHeight-ConcSlab(i)
!END DO
!DO j=1,Np
!       IF(Coord(j).LE.CurrentHeight) THEN
!               Conc(Nx)=Conc(Nx)+1.0D0
!       END IF
!END DO

!CONC AND VISC CALCS
!DO i = 0,Nx
!       Conc(i)=Conc(i)*(4.0D0/3.0D0)*Pi/(BoxSize(1)*BoxSize(2)*ConcSlab(i))
!       Visc(i)=0.001D0*eta_0*EXP(alpha*Conc(i)*ConcCorrection)
!       Visc(i)=8.56590D-4
!END DO

!OPEN(UNIT = 96, FILE = "Visc.dat")
!OPEN(UNIT = 95, FILE = "X.dat")
!DO i=0,Nx
!       WRITE(95,*) x(i)
```

```
!       WRITE(96,*) Visc(i)
!END DO
!CLOSE(UNIT = 96)
!CLOSE(UNIT = 95)

AC=0.0D0
bC=0.0D0
DO i=0,Nx-1
        AC(2*i+1,4*i+1)=1.0D0
        AC(2*i+1,4*i+2)=x(i)*RadiusP*HalfHeight
        AC(2*i+1,4*i+3)=(x(i)*RadiusP*HalfHeight)**2.0D0
        AC(2*i+1,4*i+4)=(x(i)*RadiusP*HalfHeight)**3.0D0
        AC(2*i+2,4*i+1)=1.0D0
        AC(2*i+2,4*i+2)=x(i+1)*RadiusP*HalfHeight
        AC(2*i+2,4*i+3)=(x(i+1)*RadiusP*HalfHeight)**2.0D0
        AC(2*i+2,4*i+4)=(x(i+1)*RadiusP*HalfHeight)**3.0D0
END DO
DO i=1,Nx-1
        AC(2*(Nx)+2*i-1,4*(i-1)+2)=1.0D0
        AC(2*(Nx)+2*i-1,4*(i-1)+3)=2.0D0*x(i)*RadiusP*HalfHeight
        AC(2*(Nx)+2*i-1,4*(i-1)+4)=3.0D0*(x(i)*RadiusP*HalfHeight)**2.0D0
        AC(2*(Nx)+2*i-1,4*(i-1)+6)=-1.0D0
        AC(2*(Nx)+2*i-1,4*(i-1)+7)=-2.0D0*x(i)*RadiusP*HalfHeight
        AC(2*(Nx)+2*i-1,4*(i-1)+8)=-3.0D0*(x(i)*RadiusP*HalfHeight)**2.0D0
        AC(2*(Nx)+2*i,4*(i-1)+3)=2.0D0
        AC(2*(Nx)+2*i,4*(i-1)+4)=6.0D0*x(i)*RadiusP*HalfHeight
        AC(2*(Nx)+2*i,4*(i-1)+7)=-2.0D0
        AC(2*(Nx)+2*i,4*(i-1)+8)=-6.0D0*x(i)*RadiusP*HalfHeight
END DO
AC(4*(Nx)-1,3)=2.0D0
AC(4*(Nx)-1,4)=6.0D0*x(0)*RadiusP*HalfHeight
AC(4*(Nx),4*(Nx)-1)=2.0D0
AC(4*(Nx),4*(Nx))=6.0D0*x(Nx)*RadiusP*HalfHeight

bC(1)=Conc(0)
DO i=1,Nx-1
        bC(2*i)=Conc(i)
        bC(2*i+1)=Conc(i)
END DO
bC(2*Nx)=Conc(Nx)

!SOLVE LEAST-SQUARES
CALL DGESV(4*Nx,1,AC,4*Nx,IpivC,bC,4*Nx,InfoC)

IF(InfoC.NE.0) THEN
        IF(InfoC.LT.0) THEN
```

```
            WRITE(*,*) "Illegal Call to Solver in Concentration, Input Error", InfoC
            CALL MPI_ABORT(MPI_COMM_WORLD,3,ierr)
        ELSEIF(InfoC.GT.0) THEN
            WRITE(*,*) "Singlular System In Concentration Solution Call", InfoC
            CALL MPI_ABORT(MPI_COMM_WORLD,4,ierr)
        END IF
END IF

ConcCoeff=bC
OPEN(UNIT = 96, FILE = "ConcCoeff.dat")
DO i = 1,4*Nx
        WRITE(96,"(E16.10)") ConcCoeff(i)
END DO
CLOSE(UNIT = 96)

!OLD CONCENTRATION GRADIENT CALCULATIONS
!DO i=1,Nx-1
!       ConcGrad(i)=(Conc(i-1)-Conc(i+1))*ConcCorrection/((x(i-1)-
x(i+1))*RadiusP*HalfHeight)
!       WRITE(*,*) ConcGrad(i)
!END DO

!NEW CONCENTRATION GRADIENT CALCULATIONS
!DO i=0,NConc
!       DO j=1,Nx-1
!
        ConcGrad(j)=ConcGrad(j)+DBLE(i)*ConcCoeff(i+1)*(x(j)*RadiusP*HalfHeight
)**DBLE((i-1))
!       END DO
!END DO

DO i=1,Nx-1
        ConcGrad(i)=ConcCoeff(4*i+2)+2.0D0*ConcCoeff(4*i+3)*x(i)*RadiusP*HalfH
eight+&
        3.0D0*ConcCoeff(4*i+4)*(x(i)*RadiusP*HalfHeight)**2.0D0
END DO

!OPEN(UNIT = 96, FILE = "ConcGrad.dat")
!OPEN(UNIT = 95, FILE = "X.dat")
!DO i=1,Nx-1
!       WRITE(95,*) x(i)
!       WRITE(96,*) ConcGrad(i)
!END DO
!CLOSE(UNIT = 96)
!CLOSE(UNIT = 95)
```

```
!SETUP SPECTRAL SYSTEM
DO i=1,Nx-1
        DO j=0,Nx1
                A1(i,j+1)=Visc(i)*Tx2(i,j)/((RadiusP*HalfHeight)**2.0D0)
!CORRECTED THE SCALING BY DIVIDING BY THE CHAMBER HALF HEIGHT
SQUARED
                A2(i,j+1)=alpha*Visc(i)*ConcGrad(i)*Tx1(i,j)/(RadiusP*HalfHeight)
!CORRECTED THE SCALING BY DIVIDING BY THE CHAMBER HALF HEIGHT
        END DO
        b(i)=DelP
END DO
A=A1+A2
!ADD BOUNDARY CONDITIONS
DO j=1,Nx1+1
        A(Nx,j)=1.0D0
        A(Nx+1,j)=(-1.0D0)**(j-1)
END DO
b(Nx)=0.0D0; b(Nx+1)=0.0D0


!OPEN(UNIT= 98, FILE = "A1.dat")
!OPEN(UNIT= 99, FILE = "A2.dat")
!DO i = 1,Nx-1
!       DO j = 1,Nx+1
!               WRITE(98, "(1X,F16.2)", ADVANCE ="NO") A1(i,j)
!               WRITE(99, "(1X,F16.2)", ADVANCE ="NO") A2(i,j)
!       END DO
!       WRITE(98,*)
!       WRITE(99,*)
!END DO
!CLOSE(UNIT = 98)
!CLOSE(UNIT = 99)

!SOLVE SYSTEM FOR VELOCITY COEFFICIENTS
CALL DGESV(Nx1+1,1,A,Nx1+1,Ipiv,b,Nx1+1,Info)
!CALL LUDCMP(Nx+1,A,Ipiv)
!CALL LUBKSB(Nx+1,A,Ipiv,b)
VelCoeff=b

OPEN(UNIT = 97, FILE = "VelCoeff.dat")
DO i=1,Nx+1
        WRITE(97,*) VelCoeff(i)
END DO
CLOSE(UNIT = 97)

!WRITE(*,*) "Velocity Matrix Solved!"
```

```
IF(Info.NE.0) THEN
        IF(Info.LT.0) THEN
                WRITE(*,*) "Illegal Call to Solver in Velocity, Input Error", Info
                CALL MPI_ABORT(MPI_COMM_WORLD,5,ierr)
        ELSEIF(Info.GT.0) THEN
                WRITE(*,*) "Singlular System In Velocity Solution Call", Info
                CALL MPI_ABORT(MPI_COMM_WORLD,6,ierr)
        END IF
END IF

!NEWTON'S METHOD TO FIND MAX VELOCITY
DerVel=0.0D0
Der2Vel=0.0D0
DO i=0,Nx1
        T(i)=COS(i*ACOS(MaxVelLoc))
END DO
T1(0)=0.0D0;  T1(1)=1.0D0
DO i=1,Nx1-1
        T1(i+1) = 2.0D0*MaxVelLoc*T1(i) + 2.0D0*T(i) - T1(i-1)
END DO
DO i=1,Nx1+1
        DerVel=DerVel+VelCoeff(i)*T1(i-1)
END DO
T2(0)=0.0D0;  T2(1)=0.0D0;  T2(2)=4.0D0
DO i=1,Nx1-1
        T2(i+1) = 2.0D0*MaxVelLoc*T2(i) + 4.0D0*T1(i) - T2(i-1)
END DO
DO i=1,Nx1+1
        Der2Vel=Der2Vel+VelCoeff(i)*T2(i-1)
END DO

Tol=ABS(DerVel)
OldLoc=MaxVelLoc
Iter=1
DO WHILE((Tol.GT.(1.0D-2)).AND.(Iter.LE.MaxIter).AND.(ABS(Der2Vel).GE.(1.0D-
8)))
        MaxVelLoc=MaxVelLoc-DerVel/Der2Vel
        DO i=0,Nx1
                T(i)=COS(i*ACOS(MaxVelLoc))
        END DO
        T1(0)=0.0D0;  T1(1)=1.0D0
        DO i=1,Nx1-1
                T1(i+1) = 2.0D0*MaxVelLoc*T1(i) + 2.0D0*T(i) - T1(i-1)
        END DO
        DO i=1,Nx1+1
```

```
                DerVel=DerVel+VelCoeff(i)*T1(i-1)
        END DO
        T2(0)=0.0D0;  T2(1)=0.0D0;  T2(2)=4.0D0
        DO i=1,Nx1-1
                T2(i+1) = 2.0D0*MaxVelLoc*T2(i) + 4.0D0*T1(i) - T2(i-1)
        END DO
        DO i=1,Nx1+1
                Der2Vel=Der2Vel+VelCoeff(i)*T2(i-1)
        END DO
        Tol=ABS(DerVel)
        Iter=Iter+1
END DO
IF((Iter.GT.MaxIter).OR.(ABS(Der2Vel).LT.(1.0D-8))) THEN
        MaxVelLoc=0.0D0
        WRITE(*,*) "Could not find Max Velocity!"
END IF
!MaxVelLoc=0.0D0

MaxVel=0.0D0
DO i=0,Nx1
        T(i)=COS(i*ACOS(MaxVelLoc))
        MaxVel=MaxVel+VelCoeff(i+1)*T(i)
END DO
IF(MaxVel.LT.(0.0D0)) THEN
        WRITE(*,*) "MaxVel less than zero!"
        CALL MPI_ABORT(MPI_COMM_WORLD,7,ierr)
END IF
!WRITE(*,*) MaxVel
!WRITE(*,*) "Velocity Profile Subroutine Complete!"

END SUBROUTINE VelocityCalc

!=================================================================
SUBROUTINE VelViscShear(Nx1,Location,VelCoeff,Vel,ConcCoeff,eta_0,&
        alpha,ConcCorrection,Local_Visc,RadiusP,HalfHeight,&
        Shear,x)
!=================================================================
!
!This subroutine returns the local flow velocity, the local
!viscosity, and the local shear for each particle.
!
IMPLICIT NONE
INTEGER                                 ::      Nx1, i, Section
DOUBLE PRECISION                                ::      Location, Vel, eta_0, alpha,
ConcCorrection, Local_Visc, Shear, C, RadiusP, HalfHeight!, ConcSlope
DOUBLE PRECISION,       DIMENSION(1:Nx1+1)      ::      VelCoeff
```

```
DOUBLE PRECISION,    DIMENSION(1:4*Nx1)    ::    ConcCoeff
DOUBLE PRECISION,    DIMENSION(0:Nx1) ::    x
DOUBLE PRECISION,    DIMENSION(0:Nx1) ::    T, T1

!ROUTINE FOR VELCOITY COMPUTATION FOR ANY PARTICLE
Vel=0.0D0
DO i=0,Nx1
        T(i)=COS(i*ACOS(Location))
        Vel=Vel+VelCoeff(i+1)*T(i)
END DO

!OLD ROUTINE TO RETURN VISCOITY IN PARTICLE SLAB ZONE
Section=-1
i=0
DO WHILE(Section.LT.0)
        IF((Location.LE.x(i)).AND.(Location.GT.x(i+1))) THEN
                Section=i
        END IF
        i=i+1
        IF(i.GT.Nx1-1) THEN
                Section=i-1
!               WRITE(*,*) "ERROR!", Location
!               STOP
        END IF
END DO
!ConcSlope=(Conc(Section)-Conc(Section+1))/(x(Section)-x(Section+1))
C=ConcCoeff(4*Section+1)+ConcCoeff(4*Section+2)*Location*RadiusP*HalfHeight+
ConcCoeff(4*Section+3)*(Location*RadiusP*HalfHeight)**2.0D0+&
        ConcCoeff(4*Section+4)*(Location*RadiusP*HalfHeight)**3.0D0
!Local_Visc=0.001D0*eta_0*EXP(alpha*C*ConcCorrection)
!WRITE(*,*) C, ConcCorrection, alpha, eta_0
!WRITE(*,*) Local_Visc

!NEW ROUTINE TO RETUTN VISCOSITY
!C=0.0
!DO i=1,NConc+1
!       C=C+ConcCoeff(i)*(Location*RadiusP*HalfHeight)**(i-1)
!END DO
Local_Visc=0.001D0*eta_0*EXP(alpha*C*ConcCorrection)

!ROUTINE FOR SHEAR RATE FOR ANY PARTICLE
Shear=0.0D0
T1(0)=0.0D0;  T1(1)=1.0D0
DO i=1,Nx1-1
        T1(i+1) = 2.0D0*Location*T1(i) + 2.0D0*T(i) - T1(i-1)
END DO
```

```
DO i=1,Nx1+1
        Shear=Shear+(VelCoeff(i)*T1(i-1))/(RadiusP*HalfHeight)
END DO

END SUBROUTINE VelViscShear

!=====================================================================
SUBROUTINE Cheby(Nx,Nx1,x,Tx,Tx1,Tx2,ConcSlab,HalfHeight)
!=====================================================================
! file:                 cheby.f90
! written by:           Laura Campo
! created:              07/09/2007
! modified by:          Paul Boyle
! modified:             05/22/2008
!
! caluclates the chebyshev polynomials and their first through fourth derivatives in x.
!
! requires call from main program:
!
! vectors and matrices *must* be allocated to the correct dimensions in the main program
before being
! passed to and modified by this subroutine
!
! GLOSSARY:
!       i               -       spatial index in x
!       L               -       chebyshev polynomial term index in x
!       Nx              -       # of G-L collocation points in x
!       Nx1             -       # of terms in series (related to Nx)
!       x               -       vector of collocation points in x and y directions
!       Tx, Tx1, Tx2    -       cheby polynomials & 1st 2 derivatives in x (at collocation
pts)
!
! structure of Tx, Tx1, Tx2
!       each ROW (i) contains the 0th through Nx-th polynomials (or derivatives)
evaluated at a single point x(i)
!       each COLUMN (L)  contains the Lth polynomial (or derivative of it) evaluated
at each point x(0) to x(Nx)

IMPLICIT NONE
INCLUDE "HSConst.h"
INTEGER                                 ::      i, L, NT
INTEGER                                 ::      Nx, Nx1
DOUBLE PRECISION,       DIMENSION(0:Nx)                 ::      x

DOUBLE PRECISION,       DIMENSION(0:Nx,0:Nx1)           ::      Tx, Tx1, Tx2
DOUBLE PRECISION,       DIMENSION(0:Nx)                 ::      ConcSlab
```

```
DOUBLE PRECISION                                    ::      CurrentHeight,
HalfHeight

!-------- setup collocation grid and evenly spaced grid ---------
NT = (Nx1+1)

x = (/ (COS(i*1.0D0*Pi/Nx), i = 0,Nx) /)            ! collocation points (ordered from 1 -
> -1)

!20 format (1x, 3A10)
!21 format (1x, 3I10)
!22 format (1x, F24.16)
!open(unit = 2, file = "parameters.txt", status = "replace")
!write(2,FMT = 20) "Nx", "Nx1", "NT"
!write(2,FMT = 21) Nx, Nx1, NT
!write(2,*)
!write(2,FMT = 20) "x = "
!write(2,FMT = 22) x;          write(2,*)

!-------- build cheby polys & 1st - 4th derivatives for G-L grid points in X ---------

DO i = 0,Nx
        ! for each x  (row 0 --> x=1, row N --> x=0)
        DO L = 0,Nx1                                                    ! for
each polynomial  (0 --> T0, 1 --> T1, 2 --> T2, etc)
                Tx(i,L) = COS(1.0D0*DBLE(L*i)*Pi/DBLE(Nx))
        ! Lth polynomial at x(i)
        END DO
END DO

Tx1(:,0) = 0.0D0;               Tx1(:,1) = 1.0D0                ! initialize
derivative matrices
Tx2(:,0:1) = 0.0D0;         Tx2(:,2) = 4.0D0
!Tx3(:,0:2) = 0.0D0;        Tx3(:,3) = 24.0D0
!Tx4(:,0:3) = 0.0D0;        Tx4(:,4) = 192.0D0

DO i = 0,Nx
        DO L = 1,Nx1-1
                Tx1(i,L+1) = 2.0D0*x(i)*Tx1(i,L) + 2.0D0*Tx(i,L) - Tx1(i,L-1)    ! 1st
der. Lth polynomial at x(i)
                Tx2(i,L+1) = 2.0D0*x(i)*Tx2(i,L) + 4.0D0*Tx1(i,L) - Tx2(i,L-1)  ! 2nd
der. Lth polynomial at x(i)
!               Tx3(i,L+1) = 2.0D0*x(i)*Tx3(i,L) + 6.0D0*Tx2(i,L) - Tx3(i,L-1)  ! 3rd
der. Lth polynomial at x(i)
!               Tx4(i,L+1) = 2.0D0*x(i)*Tx4(i,L) + 8.0D0*Tx3(i,L) - Tx4(i,L-1)  ! 4th
der. Lth polynomial at x(i)
```

```
        END DO
END DO


!ConcSlab(0)=0.5D0*(x(0)-x(1))
ConcSlab(0)=0.0D0
ConcSlab(Nx)=ConcSlab(0)
CurrentHeight=x(0)-ConcSlab(0)
IF(MOD(Nx+1,2)==0) THEN
        DO i = 1,((Nx-1)/2-1)
                ConcSlab(i)=2.0D0*(CurrentHeight-x(i))
                ConcSlab(Nx-i)=ConcSlab(i)
                CurrentHeight=CurrentHeight-ConcSlab(i)
        END DO
        ConcSlab((Nx-1)/2)=CurrentHeight
        ConcSlab((Nx-1)/2+1)=ConcSlab((Nx-1)/2)
ELSE
        DO i=1,(Nx/2-1)
                ConcSlab(i)=2.0D0*(CurrentHeight-x(i))
                ConcSlab(Nx-i)=ConcSlab(i)
                CurrentHeight=CurrentHeight-ConcSlab(i)
        END DO
        ConcSlab(Nx/2)=2.0D0*CurrentHeight
END IF
ConcSlab=ConcSlab*HalfHeight

RETURN
END SUBROUTINE Cheby

!=============================================================
SUBROUTINE PressDrive(Reynolds,Viscosity,HalfHeight,&
        RadiusP,Density,DelP)
!=============================================================
!
!The subroutine cacluates the pressure drop to be used in the
!"VelocityCalc" subroutine, based on contants and the average
!flow Reynolds number.
!
IMPLICIT NONE
DOUBLE PRECISION        ::      Reynolds, Viscosity, HalfHeight, RadiusP, Density,
DelP
!DOUBLE PRECISION       ::      MaxVel

DelP = -1.5D0*Reynolds*Viscosity**2.0D0/(Density*(HalfHeight*RadiusP)**3.0D0)
!MaxVel= -1.0D0/(2.0D0*Viscosity)*DelP*(HalfHeight*RadiusP)**2.0D0
!WRITE(*,*) "MaxVel: ", MaxVel
```

```
END SUBROUTINE PressDrive

!=============================================================
SUBROUTINE ConcCorrectFactor(BoxSize,NP,VolFrac,&
        ParticleVolPerMass,RadiusP,ConcCorrection)
!=============================================================
!
!Concentration correction factor calculation.
!

IMPLICIT NONE
INCLUDE "HSConst.h"
INTEGER                         ::      NP
DOUBLE PRECISION                        ::      VolFrac, ParticleVolPerMass,
RadiusP, ConcCorrection
DOUBLE PRECISION, DIMENSION(3)  ::      BoxSize
DOUBLE PRECISION                        ::      ParticleVolume, ParticleMass,
ParticleConc

ParticleVolume = (4.0D0/3.0D0)*RadiusP**3.0D0*Pi*DBLE(NP)*1.0D6 !in mL
ParticleMass = ParticleVolume/ParticleVolPerMass !in grams
ParticleConc =
ParticleMass/(BoxSize(1)*BoxSize(2)*BoxSize(3)*RadiusP**3.0D0*1.0D3) !in g/L
!WRITE(*,*) ParticleConc
ConcCorrection = ParticleConc/VolFrac
!WRITE(*,*) ConcCorrection

END SUBROUTINE ConcCorrectFactor

!================================
!SUBROUTINE StepEnergy(Sphere,NP,BoxSize,HamSS,HamPS,SSCutoff)
!================================
!
! This subroutine is to assign the Energy
! values of all the Particles
!
!IMPLICIT NONE
!INCLUDE "HSTypes.h"
!TYPE(particle), DIMENSION(*)   ::      Sphere
!DOUBLE PRECISION, DIMENSION(*)         ::      BoxSize
!INTEGER                        ::      NP
!INTEGER                        ::      i, j
!DOUBLE PRECISION                       ::      dist, EnSS, EnPS, HamSS,
HamPS, SSCutoff, Height, OppHeight
!DOUBLE PRECISION, DIMENSION(3)         ::      DistIJ
```

```
!
!DO i = 1, NP
!	Sphere(i)%Energy = 0.0D0
!	DO j = 1, NP
!		IF(j.NE.i) THEN
!			DistIJ = Sphere(i)%Coord - Sphere(j)%Coord
!			DistIJ(1) = DistIJ(1) - ANINT(DistIJ(1)/BoxSize(1))*BoxSize(1)
!			DistIJ(2) = DistIJ(2) - ANINT(DistIJ(2)/BoxSize(2))*BoxSize(2)

!			dist = SQRT(DistIJ(1)**2.0D0 + DistIJ(2)**2.0D0 +
DistIJ(3)**2.0D0)
!			IF dist.LT.SSCutoff THEN
!				Sphere(i)%Energy = Sphere(i)%Energy - HamSS / 3.0D0 *
(1.0D0 / (dist**2.0D0 - 4.0D0) + 1.0D0/dist**2.0D0 + LOG(1.0D0 - 4.0D0 /
dist**2.0D0))
!			ELSE
!				Sphere(i)%Energy = Sphere(i)%Energy - HamSS * (1.0D0
/ dist**6.0D0) * (16.0D0 / 9.0D0)
!			END IF
!		END IF
!	END DO
!	Height = BoxSize(3) / 2.0D0 - ABS(Sphere(i)%Coord(3)) - 1.0D0
!	OppHeight = BoxSize(3) / 2.0D0 + ABS(Sphere(i)%Coord(3)) - 1.0D0
!	Sphere(i)%Energy = Sphere(i)%Energy - HamPS / 6.0D0 * (1.0D0 / Height +
1.0D0 / (2.0D0 + Height) + LOG(Height / (2.0D0 + Height)))
!	Sphere(i)%Energy = Sphere(i)%Energy - HamPS / 6.0D0 * (1.0D0 / OppHeight +
1.0D0 / (2.0D0 + OppHeight) + LOG(OppHeight / (2.0D0 + OppHeight)))
!END DO
!
!END SUBROUTINE StepEnergy

!==============================
!SUBROUTINE
ParticleEnergy(Sphere,RecvBuf,BoxSize,HamSS,HamPS,SS_VDWCutoff,i,Energy,Radi
usP,Eps_r,C_0,Beta,Electron,Valence,ElecConc,VDWFlag,VDWCutoff,SS_EDLCutoff,
WallZeta,WallValence,ElecValence,Rank)
!==============================
!
!IMPLICIT NONE
!INCLUDE "HSTypes.h"
!INCLUDE "HSConst.h"
!INCLUDE "/opt/apps/openmpi/1.3.3-intel/include/mpif.h"
!TYPE(particle), DIMENSION(*)	::	Sphere
!DOUBLE PRECISION, DIMENSION(*)		::	BoxSize
!INTEGER, DIMENSION(*)			::	RecvBuf
!INTEGER			::	i, j, Rank, jmin, jmax
```

```
!DOUBLE PRECISION                                   ::      Gam, GamWall
!DOUBLE PRECISION                                   ::      Energy, Eps_r, C_0, Eps_0,
Electron, Valence, Debye, ElecConc, Beta, WallZeta, WallValence, ElecValence
!DOUBLE PRECISION                                   ::      dist, VDWEng, HamSS,
HamPS, SS_VDWCutoff, Height, OppHeight, EDLEng, RadiusP
!DOUBLE PRECISION                                   ::      SS_EDLCutoff
!DOUBLE PRECISION, DIMENSION(3)                     ::      DistIJ
!DOUBLE PRECISION                                   ::      VDWCutoff
!LOGICAL                            ::      VDWFlag
!DOUBLE PRECISION                           ::      EDLWall,VDWWall

!WRITE(*,*) Rank, RecvBuf(1), RecvBuf(2)
!VDWFlag = .FALSE.
!Eps_0 = 1.0D0 / (4.0D0 * Pi * 1.0D-7 * C_0**2.0D0)
!PartConc = NP / (BoxSize(1) * BoxSize(2) * BoxSize(3) * RadiusP**3.0D0)
!Debye = SQRT(2.0D0 *  PartConc * Electron**2.0D0 * Valence**2.0D0 * Beta /
(Eps_0 * Eps_r))
!Debye = SQRT(2.0D0 * 1.0D-3 * ElecConc * Avogadro * Electron**2.0D0 *
ElecValence**2.0D0 * Beta / (Eps_0 * Eps_r))
!VDWEng = 0.0D0
!EDLEng = 0.0D0
!Gam = TANH(Valence * Electron * Sphere(i)%ZetaPot * Beta / 4.0D0)
!GamWall = TANH(WallValence * Electron * WallZeta * Beta / 4.0D0)
!jmin=RecvBuf(1)
!jmax=RecvBuf(2)
!DO j = jmin,jmax
!       IF(j.NE.i) THEN
!               DistIJ = Sphere(i)%Coord - Sphere(j)%Coord
!               DistIJ(1) = DistIJ(1) - ANINT(DistIJ(1)/BoxSize(1))*BoxSize(1)
!               DistIJ(2) = DistIJ(2) - ANINT(DistIJ(2)/BoxSize(2))*BoxSize(2)

!               dist = SQRT(DistIJ(1)**2.0D0 + DistIJ(2)**2.0D0 + DistIJ(3)**2.0D0)
!               IF (dist.LE.(2.0D0+VDWCutoff)) THEN
!                       VDWFlag  = .TRUE.
               !WRITE(*,*) dist, i, j, Rank
!               END IF
!               IF (dist.LT.SS_VDWCutoff) THEN
!                       VDWEng = VDWEng - HamSS / 3.0D0 * (1.0D0 / (dist**2.0D0 -
4.0D0) + 1.0D0/dist**2.0D0 + LOG(1.0D0 - 4.0D0 / dist**2.0D0))
!               ELSE
!                       VDWEng = VDWEng - HamSS * (1.0D0 / dist**6.0D0) *
(16.0D0 / 9.0D0)
!               END IF
!               IF (dist.LT.SS_EDLCutoff) THEN
```

```
!                       EDLEng = EDLEng + 32.0D0 * RadiusP * Pi * Gam**2.0D0 *
Eps_r * Eps_0 * 1.0D0 / Beta**2.0D0 / (Electron**2.0D0 * Valence**2.0D0) *
EXP(Debye * RadiusP * (2.0D0 - dist))
!               ELSE
!                       EDLEng = EDLEng + 64.0D0 * RadiusP * Pi * Gam**2.0D0 *
Eps_r * Eps_0 * 1.0D0 / Beta**2.0D0 / (Electron**2.0D0 * Valence**2.0D0) *
EXP(Debye * RadiusP * (2.0D0 - dist)) / dist
!               ENDIF
                !EDLEng = EDLEng + Valence**2.0D0 * Electron**2.0D0 / (4.0D0 * Pi
* Eps_0 * Eps_r) * EXP(Debye * RadiusP * (2.0D0 - dist)) / (RadiusP * dist * (1.0D0 +
Debye * RadiusP)**2.0D0)
!       END IF
!END DO
!IF (Rank.EQ.1) THEN
  !Height = BoxSize(3) / 2.0D0 + Sphere(i)%Coord(3) - 1.0D0
!  Height = BoxSize(3) / 2.0D0 - ABS(Sphere(i)%Coord(3)) - 1.0D0
!  OppHeight = BoxSize(3) / 2.0D0 + ABS(Sphere(i)%Coord(3)) - 1.0D0
!  VDWWall=- HamPS / 6.0D0 * (1.0D0 / Height + 1.0D0 / (2.0D0 + Height) +
LOG(Height / (2.0D0 + Height)))
!  WRITE(*,*) VDWWall

!  VDWEng = VDWEng - HamPS / 6.0D0 * (1.0D0 / Height + 1.0D0 / (2.0D0 + Height)
+ LOG(Height / (2.0D0 + Height)))
!  VDWEng = VDWEng - HamPS / 6.0D0 * (1.0D0 / OppHeight + 1.0D0 / (2.0D0 +
OppHeight) + LOG(OppHeight / (2.0D0 + OppHeight)))

!  EDLWall=64.0D0 * RadiusP * Pi * Gam * GamWall * Eps_r * Eps_0 * 1.0D0 /
Beta**2.0D0 / (Electron**2.0D0 * Valence**2.0D0) * EXP(Debye * RadiusP * (2.0D0 -
Height))
!  WRITE(*,*) EDLWall

!  EDLEng = EDLEng + 64.0D0 * RadiusP * Pi * Gam * GamWall * Eps_r * Eps_0 *
1.0D0 / Beta**2.0D0 / (Electron**2.0D0 * Valence**2.0D0) * EXP(Debye * RadiusP *
(2.0D0 - Height))
!  EDLEng = EDLEng + 64.0D0 * RadiusP * Pi * Gam * GamWall * Eps_r * Eps_0 *
1.0D0 / Beta**2.0D0 / (Electron**2.0D0 * Valence**2.0D0) * EXP(Debye * RadiusP *
(2.0D0 - OppHeight))
!END IF

!Energy = VDWEng + EDLEng

!END SUBROUTINE ParticleEnergy

!===================================
SUBROUTINE
PPEnergy(Sphere,Part1,Part2,BoxSize,HamSS,SS_VDWCutoff,Energy,RadiusP,Eps_r,B
```

```
eta,Electron,Valence,ElecConc,VDWFlag,VDWCutoff,SS_EDLCutoff,ElecValence,ED
LChoice,Debye,Rank)
!================================
!
IMPLICIT NONE
INCLUDE "HSTypes.h"
INCLUDE "HSConst.h"
INCLUDE "/opt/apps/openmpi/1.3.3-intel/include/mpif.h"
TYPE(particle), DIMENSION(*)    ::      Sphere
DOUBLE PRECISION, DIMENSION(*)          ::      BoxSize
INTEGER                 ::      Part1, Part2, EDLChoice
INTEGER                 ::      Rank
DOUBLE PRECISION                        ::      Gam
!DOUBLE PRECISION                       ::      Eps_0, C_0
DOUBLE PRECISION                        ::      Energy, Eps_r, Electron,
Valence, Debye, ElecConc, Beta, ElecValence
DOUBLE PRECISION                        ::      dist, VDWEng, HamSS,
SS_VDWCutoff, Height, OppHeight, EDLEng, RadiusP
DOUBLE PRECISION                        ::      SS_EDLCutoff
DOUBLE PRECISION                        ::      ReducedDist
DOUBLE PRECISION, DIMENSION(3)          ::      DistIJ
DOUBLE PRECISION                        ::      VDWCutoff
LOGICAL                 ::      VDWFlag

VDWFlag = .FALSE.
!Eps_0 = 1.0D0 / (4.0D0 * Pi * 1.0D-7 * C_0**2.0D0)
!Debye = SQRT(2.0D0 * 1.0D3 * ElecConc * Avogadro * Electron**2.0D0 *
ElecValence**2.0D0 * Beta / (Eps_0 * Eps_r))
VDWEng = 0.0D0
EDLEng = 0.0D0
DistIJ = Sphere(Part1)%Coord - Sphere(Part2)%Coord
DistIJ(1) = DistIJ(1) - ANINT(DistIJ(1)/BoxSize(1))*BoxSize(1)
DistIJ(2) = DistIJ(2) - ANINT(DistIJ(2)/BoxSize(2))*BoxSize(2)
dist = SQRT(DistIJ(1)**2.0D0 + DistIJ(2)**2.0D0 + DistIJ(3)**2.0D0)
IF (dist.LE.(2.0D0+VDWCutoff)) THEN
        VDWFlag  = .TRUE.
        WRITE(*,*) dist, Part1, Part2, Rank
END IF
IF (dist.LT.SS_VDWCutoff) THEN
        VDWEng = VDWEng - HamSS / 3.0D0 * (1.0D0 / (dist**2.0D0 - 4.0D0) +
1.0D0/dist**2.0D0 + LOG(1.0D0 - 4.0D0 / dist**2.0D0))
ELSE
        VDWEng = VDWEng - HamSS * (1.0D0 / dist**6.0D0) * (16.0D0 / 9.0D0)
END IF
ReducedDist=dist - ((2.0D0/Debye)/RadiusP)
IF (EDLChoice.EQ.1) THEN
```

```
        Gam = TANH(Valence * Electron * Sphere(Part1)%ZetaPot * Beta / 4.0D0)
        IF (ReducedDist.LT.SS_EDLCutoff) THEN
                EDLEng = EDLEng + 32.0D0 * RadiusP * Pi * Gam**2.0D0 * Eps_r *
Eps_0 * 1.0D0 / Beta**2.0D0 / (Electron**2.0D0 * Valence**2.0D0) * EXP(Debye *
RadiusP * (2.0D0 - ReducedDist))
        ELSE
                EDLEng = EDLEng + 64.0D0 * RadiusP * Pi * Gam**2.0D0 * Eps_r *
Eps_0 * 1.0D0 / Beta**2.0D0 / (Electron**2.0D0 * Valence**2.0D0) * EXP(Debye *
RadiusP * (2.0D0 - ReducedDist)) / ReducedDist
        ENDIFELSE
        EDLEng = EDLEng + Pi * Eps_r * Eps_0 * RadiusP * (dist - 1.0D0) / (2.0D0 *
dist) *&
                (((Sphere(Part1)%ZetaPot)**2.0D0 + (Sphere(Part2)%ZetaPot)**2.0D0 +
2.0D0 * (Sphere(Part1)%ZetaPot) * (Sphere(Part2)%ZetaPot)) * LOG(1.0D0 +
SQRT(EXP(Debye * RadiusP * (2.0D0 - dist))) / (dist - 1.0D0)) +&
                ((Sphere(Part1)%ZetaPot)**2.0D0 + (Sphere(Part2)%ZetaPot)**2.0D0 -
2.0D0 * (Sphere(Part1)%ZetaPot) * (Sphere(Part2)%ZetaPot)) * LOG(1.0D0 -
SQRT(EXP(Debye * RadiusP * (2.0D0 - dist))) / (dist - 1.0D0)))
ENDIF
Energy = VDWEng + EDLEng

END SUBROUTINE PPEnergy

!================================
SUBROUTINE
PWEnergy(Sphere,BoxSize,HamPS,Energy,RadiusP,Eps_r,Beta,Electron,Valence,ElecC
onc,WallZeta,WallValence,ElecValence,Debye,Rank)
!================================
!
IMPLICIT NONE
INCLUDE "HSTypes.h"
INCLUDE "HSConst.h"
INCLUDE "/opt/apps/openmpi/1.3.3-intel/include/mpif.h"
TYPE(particle)          ::      Sphere
DOUBLE PRECISION, DIMENSION(*)          ::      BoxSize
INTEGER                 ::      Rank
DOUBLE PRECISION                        ::      Gam, GamWall
!DOUBLE PRECISION                       ::      Eps_0, C_0
DOUBLE PRECISION                        ::      Energy, Eps_r, Electron,
Valence, Debye, ElecConc, Beta, WallZeta, WallValence, ElecValence
DOUBLE PRECISION                        ::      VDWEng, HamPS, Height,
OppHeight, EDLEng, RadiusP

!Eps_0 = 1.0D0 / (4.0D0 * Pi * 1.0D-7 * C_0**2.0D0)
!Debye = SQRT(2.0D0 * 1.0D3 * ElecConc * Avogadro * Electron**2.0D0 *
ElecValence**2.0D0 * Beta / (Eps_0 * Eps_r))
```

```
Gam = TANH(Valence * Electron * Sphere%ZetaPot * Beta / 4.0D0)
GamWall = TANH(WallValence * Electron * WallZeta * Beta / 4.0D0)
Height = BoxSize(3) / 2.0D0 - ABS(Sphere%Coord(3)) - 1.0D0
OppHeight = BoxSize(3) / 2.0D0 + ABS(Sphere%Coord(3)) - 1.0D0
VDWEng=0.0D0
EDLEng=0.0D0
VDWEng = VDWEng - HamPS / 6.0D0 * (1.0D0 / Height + 1.0D0 / (2.0D0 + Height) +
LOG(Height / (2.0D0 + Height)))
VDWEng = VDWEng - HamPS / 6.0D0 * (1.0D0 / OppHeight + 1.0D0 / (2.0D0 +
OppHeight) + LOG(OppHeight / (2.0D0 + OppHeight)))

EDLEng = EDLEng + 64.0D0 * RadiusP * Pi * Gam * GamWall * Eps_r * Eps_0 *
1.0D0 / Beta**2.0D0 / (Electron**2.0D0 * Valence**2.0D0) * EXP(Debye * RadiusP *
(-Height+((2.0D0/Debye)/RadiusP)))
EDLEng = EDLEng + 64.0D0 * RadiusP * Pi * Gam * GamWall * Eps_r * Eps_0 *
1.0D0 / Beta**2.0D0 / (Electron**2.0D0 * Valence**2.0D0) * EXP(Debye * RadiusP *
(-OppHeight+((2.0D0/Debye)/RadiusP)))

Energy = VDWEng + EDLEng

END SUBROUTINE PWEnergy

!================================
RECURSIVE SUBROUTINE Cascade(Sphere,Old,ip,NP,BoxSize,VDWCutoff)
!================================
!
IMPLICIT NONE
INCLUDE "HSTypes.h"
INCLUDE "/opt/apps/openmpi/1.3.3-intel/include/mpif.h"
TYPE(particle), DIMENSION(*)     ::        Sphere, Old
INTEGER :: ip, jp, NP
INTEGER     ::        ierr
LOGICAL :: Overlap
DOUBLE PRECISION :: VDWCutoff
DOUBLE PRECISION, DIMENSION(*) :: BoxSize

DO jp=1,NP
        IF (jp.NE.ip) THEN
                CALL
CheckOverlapIJPer(Sphere(ip),Sphere(jp),BoxSize,Overlap,VDWCutoff)
                IF (Overlap==.TRUE.) THEN
                        IF (Sphere(jp)%Returned.EQ.1) THEN
                                WRITE(*,*) "FATAL Cascade Error!!!!"
                                CALL MPI_ABORT(MPI_COMM_WORLD,8,ierr)
                        ENDIF
                        Sphere(jp)%Coord=Old(jp)%Coord
```

```
                        Sphere(jp)%BadMove=1
                        Sphere(jp)%Returned=1
                        CALL Cascade(Sphere,Old,jp,NP,BoxSize,VDWCutoff)
                ENDIF
        ENDIF
ENDDO

END SUBROUTINE Cascade

!================================
SUBROUTINE InitSurface(Sphere,Surf,BoxSize,SurfPotential)
!================================
!
 IMPLICIT NONE
 INCLUDE "HSTypes.h"

 TYPE(particle), DIMENSION(*)    ::        Sphere
 DOUBLE PRECISION,          DIMENSION(*) :: BoxSize
 INTEGER :: Surf
 DOUBLE PRECISION    :: RandNum
!  DOUBLE PRECISION    :: UpDown
 INTEGER :: i, j
 LOGICAL :: Overlap, OverlapTest
 DOUBLE PRECISiON     :: SurfPotential
!
! For the first particle (index = 1)
!
 Sphere(1)%Radius=1.0D0 !Must be 1.0 for scaled equations to hold
 Sphere(1)%ZetaPot=SurfPotential

! To put particles randomly in x,y-directions on the surfaces
 CALL RANDOM_NUMBER(RandNum)
 Sphere(1)%Coord(1) = 2.0D0*( 0.5D0 - RandNum ) * ( 0.5D0*BoxSize(1) -
Sphere(1)%Radius )
 CALL RANDOM_NUMBER(RandNum)
 Sphere(1)%Coord(2) = 2.0D0*( 0.5D0 - RandNum ) * ( 0.5D0*BoxSize(2) -
Sphere(1)%Radius )
! Place first sphere on bottom
 Sphere(1)%Coord(3) = -( 0.5D0*BoxSize(3) - Sphere(1)%Radius )

!Random choice of top or bottom
!  CALL RANDOM_NUMBER(RandNum)
!  UpDown = 2.0D0*( 0.5D0 - RandNum )
!  IF (UpDown.LT.0.0D0) THEN
!        Sphere(1)%Coord(3) = -( 0.5D0*BoxSize(3) - Sphere(1)%Radius )
!  ELSE
```

```
!         Sphere(1)%Coord(3) = ( 0.5D0*BoxSize(3) - Sphere(1)%Radius )
!  ENDIF

!
! For the rest of the particles (index>1)
!
  SphereLoop:&
  DO i = 2, Surf
    Sphere(i)%Radius=1.0D0
    Sphere(i)%ZetaPot=SurfPotential
    Overlap = .TRUE.
    DO WHILE (Overlap)

      CALL RANDOM_NUMBER(RandNum)
      Sphere(i)%Coord(1) = 2.0D0*( 0.5D0-RandNum) * (0.5D0*BoxSize(1)-
Sphere(i)%Radius)
      CALL RANDOM_NUMBER(RandNum)
      Sphere(i)%Coord(2) = 2.0D0*( 0.5D0-RandNum) * (0.5D0*BoxSize(2)-
Sphere(i)%Radius)
      IF (MOD(i,2).EQ.1) THEN
          Sphere(i)%Coord(3) = -( 0.5D0*BoxSize(3) - Sphere(i)%Radius )
      ELSE
          Sphere(i)%Coord(3) = ( 0.5D0*BoxSize(3) - Sphere(i)%Radius )
      ENDIF
!Random choice of top or bottom
!      CALL RANDOM_NUMBER(RandNum)
!      UpDown = 2.0D0*( 0.5D0 - RandNum )
!      IF (UpDown.LT.0.0D0) THEN
!          Sphere(i)%Coord(3) = -( 0.5D0*BoxSize(3) - Sphere(i)%Radius )
!      ELSE
!          Sphere(i)%Coord(3) = ( 0.5D0*BoxSize(3) - Sphere(i)%Radius )
!      ENDIF

      OverlapTest = .FALSE.
      DO j = 1, i-1
        CALL CheckOverlapIJ (Sphere(i),Sphere(j),OverlapTest,0)
        IF(OverlapTest==.TRUE.) EXIT
      END DO
      Overlap = OverlapTest

    END DO

  END DO &
  SphereLoop

END SUBROUTINE InitSurface
```

```
!================================
SUBROUTINE
PWSEnergy(Sphere1,Sphere2,BoxSize,HamSS,SS_VDWCutoff,Energy,RadiusP,Eps_r,
Beta,Electron,Valence,ElecConc,VDWFlag,VDWCutoff,SS_EDLCutoff,SurfValence,El
ecValence,EDLChoice,Debye,Rank)
!================================
!
IMPLICIT NONE
INCLUDE "HSTypes.h"
INCLUDE "HSConst.h"


TYPE(particle)                          ::      Sphere1, Sphere2
DOUBLE PRECISION, DIMENSION(*)                  ::      BoxSize
INTEGER                         ::      Part1, Part2, EDLChoice
INTEGER                         ::      Rank
DOUBLE PRECISION                                ::      Gam, GamWall
!DOUBLE PRECISION                               ::      Eps_0, C_0
DOUBLE PRECISION                                ::      Energy, Eps_r, Electron,
Valence, Debye, ElecConc, Beta, SurfValence, ElecValence
DOUBLE PRECISION                                ::      dist, VDWEng, HamSS,
SS_VDWCutoff, Height, OppHeight, EDLEng, RadiusP
DOUBLE PRECISION                                ::      SS_EDLCutoff
DOUBLE PRECISION                                ::      ReducedDist
DOUBLE PRECISION, DIMENSION(3)                  ::      DistIJ
DOUBLE PRECISION                                ::      VDWCutoff
LOGICAL                         ::      VDWFlag

VDWFlag = .FALSE.
!Eps_0 = 1.0D0 / (4.0D0 * Pi * 1.0D-7 * C_0**2.0D0)
!Debye = SQRT(2.0D0 * 1.0D3 * ElecConc * Avogadro * Electron**2.0D0 *
ElecValence**2.0D0 * Beta / (Eps_0 * Eps_r))
VDWEng = 0.0D0
EDLEng = 0.0D0
DistIJ = Sphere1%Coord - Sphere2%Coord
DistIJ(1) = DistIJ(1) - ANINT(DistIJ(1)/BoxSize(1))*BoxSize(1)
DistIJ(2) = DistIJ(2) - ANINT(DistIJ(2)/BoxSize(2))*BoxSize(2)
dist = SQRT(DistIJ(1)**2.0 + DistIJ(2)**2.0 + DistIJ(3)**2.0)
IF (dist.LE.(2.0D0+VDWCutoff)) THEN
        VDWFlag  = .TRUE.
        WRITE(*,*) dist, Rank
END IF
IF (dist.LT.SS_VDWCutoff) THEN
        VDWEng = VDWEng - HamSS / 3.0D0 * (1.0D0 / (dist**2.0D0 - 4.0D0) +
1.0D0/dist**2.0D0 + LOG(1.0D0 - 4.0D0 / dist**2.0D0))
```

```
ELSE
        VDWEng = VDWEng - HamSS * (1.0D0 / dist**6.0D0) * (16.0D0 / 9.0D0)
END IF
ReducedDist=dist - ((2.0D0/Debye)/RadiusP)
IF (EDLChoice.EQ.1) THEN
        Gam = TANH(Valence * Electron * Sphere1%ZetaPot * Beta / 4.0D0)
        GamWall = TANH(SurfValence * Electron * Sphere2%ZetaPot * Beta / 4.0D0)
        IF (ReducedDist.LT.SS_EDLCutoff) THEN
                EDLEng = EDLEng + 32.0D0 * RadiusP * Pi * Gam*GamWall * Eps_r *
Eps_0 * 1.0D0 / Beta**2.0D0 / (Electron**2.0D0 * Valence**2.0D0) * EXP(Debye *
RadiusP * (2.0D0 - ReducedDist))
        ELSE
                EDLEng = EDLEng + 64.0D0 * RadiusP * Pi * Gam*GamWall * Eps_r *
Eps_0 * 1.0D0 / Beta**2.0D0 / (Electron**2.0D0 * Valence**2.0D0) * EXP(Debye *
RadiusP * (2.0D0 - ReducedDist)) / ReducedDist
        ENDIF
ELSE
        EDLEng = EDLEng + Pi * Eps_r * Eps_0 * RadiusP * (dist - 1.0D0) / (2.0D0 *
dist) *&
                (((Sphere1%ZetaPot)**2.0D0 + (Sphere2%ZetaPot)**2.0D0 + 2.0D0 *
(Sphere1%ZetaPot) * (Sphere2%ZetaPot)) * LOG(1.0D0 + SQRT(EXP(Debye *
RadiusP * (2.0D0 - dist))) / (dist - 1.0D0)) +&
                ((Sphere1%ZetaPot)**2.0D0 + (Sphere2%ZetaPot)**2.0D0 - 2.0D0 *
(Sphere1%ZetaPot) * (Sphere2%ZetaPot)) * LOG(1.0D0 - SQRT(EXP(Debye * RadiusP
* (2.0D0 - dist))) / (dist - 1.0D0)))
ENDIF

Energy = VDWEng + EDLEng

END SUBROUTINE PWSEnergy
```

HSTypes.h

```
TYPE particle         ! Defining the DATA TYPE of "particle"
  DOUBLE PRECISION, DIMENSION(3) :: Coord
  DOUBLE PRECISION :: InitialZ
  DOUBLE PRECISION :: Radius
  DOUBLE PRECISION :: ZetaPot
  DOUBLE PRECISION :: Time
  DOUBLE PRECISION :: OldEnergy
  DOUBLE PRECISION :: NewEnergy
  INTEGER          :: BadMove
  INTEGER          :: Returned
  INTEGER          :: Moves
END TYPE particle
```

HSinf.f90

```
!=== Writing the Key Information to MCRunInf file (i.e., Job____.msg file)
!
iFile = 10
OPEN(iFile,file=MCRunInf)
  WRITE(iFile,*) "====================================="
  WRITE(iFile,*) "The Number of Particles:            ", NP
  WRITE(iFile,*) "Radius of Each Particle (meter):    ", RadiusP
  WRITE(iFile,*) "Permeate Velocity (meter/sec):      ", Permeate
  WRITE(iFile,*) "Feed Volume Fraction:               ", VolFrac
  WRITE(iFile,*) "Box Length:                         ", BoxSize(1)
  WRITE(iFile,*) "Box Width :                         ", BoxSize(2)
  WRITE(iFile,*) "Box Height:                         ", BoxSize(3)
  WRITE(iFile,*) "Gap:                                ", Gap
  WRITE(iFile,*) "Particle Zeta Potental (Volts)      ", zeta
  WRITE(iFile,*) "Solvetn (water) Viscosity:          ", Viscosity
  WRITE(iFile,*) "The Number of Total Simulation Steps:", Nstep
  WRITE(iFile,*) "The Number of Pre-equilibrium Steps: ", Nequil
  WRITE(iFile,*) "The Update Interval:                ", Nupdate
  WRITE(iFile,*) "The Number of Bins in z-direction:  ", Nbin
  WRITE(iFile,*) "The File Name of This File:         ", MCRunInf
  WRITE(iFile,*) "The Inital Particle Coordinates :   ", XYZbegin
  WRITE(iFile,*) "The Final  Particle Coordinates :   ", XYZfinal
  WRITE(iFile,*) "The Vertical Density Profile, Conc: ", DensityZ
  WRITE(iFile,*) "The Flow Field Profiles       :     ", Velocity
  WRITE(iFile,*) "Reynolds Number               :     ", Reynolds
  WRITE(iFile,*) "Ven der Waals Cutoff Ratio    :     ", SS_VDWCutoff
  WRITE(iFile,*) "Processors                    :     ", Procs
  WRITE(iFile,*) "====================================="
  WRITE(iFile,*) " "
CLOSE(iFile)
```

HSout.f90

```
 !=== Writing the initial coordinates of all the particles
 !
 iFile = 11
 OPEN(iFile,file=XYZbegin)
   WRITE(iFile,*) NP ; WRITE(iFile,*) "!"
   DO ip = 1, NP
      WRITE(iFile,'(" O ",3(2X,E16.8))') ColloidIni(ip)%Coord
   END Do
 CLOSE(iFile)

 ! Writing the final coordiates of all the particles
 !
 iFile=12
 OPEN(iFile,file=XYZfinal)
   WRITE(iFile,*) NP ; WRITE(iFile,*) "!"
   DO ip = 1, NP
      WRITE(iFile,'(" O ",3(2X,E16.8))') ColloidIni(ip)%Coord
!      WRITE(iFile,*) "O", Colloid(ip)%Coord
   END Do
 CLOSE(iFile)

 ! Writing the density profiles of particles
 !
 iFile=13
 OPEN(iFile,file=DensityZ)
   DO ibin = -Nbin, Nbin
     WRITE(iFile,*) Znorm(ibin) , ConcIni(ibin) ,Conc(ibin) , ConcAvg(ibin)
   END Do
 CLOSE(iFile)

 ! Writing the Velocity Field U(z) and V(z)
 ! in x- and z-directions, respectively.
 !
!  iFile=14
!  OPEN(iFile,file=Velocity)
!    DO ibin = -Nbin, Nbin
!      WRITE(iFile,*) Znorm(ibin) , FlowXnorm(ibin)!, FlowZnorm(ibin)
!    END Do
!  CLOSE(iFile)

 iFile=15
 OPEN(iFile,file=SimulRes)
     WRITE(iFile,*) "LOG10(Permeate),  PSI_AVG,  RadiusP, NP, SResistance"
     WRITE(iFile,*) LOG10(Permeate),  PSI_AVG,  RadiusP, NP, SResistance
```

```
CLOSE(iFile)
```

HSParam.h

```
!Cubic Spline Concentration
!Parameters: February 18, 2011
!=================================================================
!
CHARACTER (LEN= 1) :: InitStructure="R"   ! "Random" or "Previous"
!
CHARACTER (LEN=60) :: Previous="MC_20071010_172031_HS_XYZ00040.xyz" !
For example
!
INTEGER ::                      iSeqFilePre = 00040
!
!=================================================================
!
DOUBLE PRECISION    :: VolFracFeed       =    1.0000D-4
! Feed Volume Fraction of a real system
DOUBLE PRECISION    :: VolFrac           =    0.1D0
! Initial Volume Fraction in the channel
!
! DOUBLE PRECISION    :: DiameterP              =      6.2600D-9
! DOUBLE PRECISION    :: DiameterP         =    0.1000D-6    !
! DOUBLE PRECISION    :: DiameterP         =    0.3000D-6    !
! DOUBLE PRECISION    :: DiameterP         =    1.000D-6    !
! DOUBLE PRECISION    :: DiameterP         =    3.000D-6    !
 DOUBLE PRECISION    :: DiameterP         =    10.00D-6    !
!

! DOUBLE PRECISION    :: Permeate          =    0.000000D0
! Mean Permeate Velocity 0
! DOUBLE PRECISION    :: Permeate          =    1.000000D-6
! Mean Permeate Velocity 1
! DOUBLE PRECISION    :: Permeate          =    5.000000D-6
! Mean Permeate Velocity 2
! DOUBLE PRECISION    :: Permeate          =    10.000000D-6
! Mean Permeate Velocity 3
! DOUBLE PRECISION    :: Permeate          =    20.000000D-6
! Mean Permeate Velocity 4
 DOUBLE PRECISION    :: Permeate          =    30.000000D-6
! Mean Permeate Velocity 5
!
!

DOUBLE PRECISION    :: DeltaSpecificDensity =    0.0D0
! (density_particle - density_fluid)/density_fluid
DOUBLE PRECISION    :: zeta              =    -30.0D-3
```

```
! Particle Zeta Potental (Volts)
!
!CHARACTER (LEN= 1) :: VolumeExpansion="Y"   ! "Y" or "y"
!
INTEGER :: NPmax         =  2100        ! The Number of Particles,
!                                 ! If NPmax = NP, canonical plus no killing
!                                 ! If NPmax > NP, Grand Canonical (creation + killing)
INTEGER :: NP           =  2100        ! The Number of Particles
INTEGER :: Nstep         =  10000         ! The Number of Simulation Steps
!
CHARACTER (LEN= 1) :: FixedBox           =  "N"
!Use a fixed height or use a fixed aspect ratio, "Y" for fixed height, "N" for fixed aspect
ratio
DOUBLE PRECISION :: FixedH =  112.8394367811528D0
!Height for 2100 particles at 10% volume fraction
INTEGER :: NboxZ         =  7           ! The ratio of Height to Length
INTEGER :: NboxX         =  3           ! The ratio of Width to Length
!
INTEGER :: Nupdate       =  10          ! The Update Interval
INTEGER :: NupdateCK     =  2
! The Update Interval of Particle compen + kill
INTEGER :: NDdataCollect  =  2          ! if 3, 1/3 of initial data won't be used
INTEGER :: NMovieInterval =  100
! The Interval of MC step for movie makeing
INTEGER :: Nbin          =  50          ! The Number of Bins in z-direction (one side)
INTEGER :: Npeek         =  10          ! The Interval Number for "bpeek" command
INTEGER :: NmovieMaxStep  =  100000         ! The Maximum MC step for movie
!
CHARACTER(1)      :: Grid =  "Grid"        ! "G" or "g" for grid, anything else for no
grid
CHARACTER (LEN=1)  :: Vid  = "Y"                !N for no video files, Y for video
!
CHARACTER (LEN=60) :: ExtraData  ! Unit 80 ! 'data/MC25_HS_Time.dat'
CHARACTER (LEN=60) :: PeekData  ! Unit 9  ! 'data/MC25_HS_Peek.msg'
CHARACTER (LEN=60) :: MCRunInf  ! Unit 10 ! 'data/MC25_HS_All_Info.dat'
CHARACTER (LEN=60) :: XYZbegin  ! Unit 11 ! 'data/MC25_HS_XYZbegin.xyz'
CHARACTER (LEN=60) :: XYZfinal  ! Unit 12 ! 'data/MC25_HS_XYZfinal.xyz'
CHARACTER (LEN=60) :: DensityZ  ! Unit 13 ! 'data/MC25_HS_DensityZ.dat'
CHARACTER (LEN=60) :: Velocity  ! Unit 14 ! 'data/MC25_HS_Velocity.dat'
CHARACTER (LEN=60) :: SimulRes  ! Unit 15 ! 'data/MC25_HS_SimulRes.dat'
CHARACTER (LEN= 1) :: Identity

DOUBLE PRECISION    :: Lambda        =    0.50D0
! The factor of Force-Bias MC
DOUBLE PRECISION    :: Viscosity     =    8.887D-4
! Water Viscosity Based on T in Beta
```

```
DOUBLE PRECISION   :: DensityFluid  =   996.63D0      ! Water Density
DOUBLE PRECISION   :: Gravity      =   9.80D0        ! Gravitational Acceleration
DOUBLE PRECISION   :: Beta         =   2.4143D+20
! (=1/kT) = 1/ (1.3806503 x 10E-23 x 300)
INTEGER :: Nequil      =   0          ! The Number of Pre-equilibrium Steps

!=====================================================================


!=====================================================================
!Velocity Solution Parameters
DOUBLE PRECISION          ::      Reynolds            =     1000.0D0
DOUBLE PRECISION          ::      Density             =      996.63D0
!Based on T in Beta, pure water
DOUBLE PRECISION          ::      alpha               =      0.0097D0
DOUBLE PRECISION          ::      eta_0               =      0.8887D0
DOUBLE PRECISION          ::      ParticleVolPerMass  =      0.789D0
!in mL/g
INTEGER          ::      Nx                  =      24
INTEGER :: EndSteps      =      100          !Steps to Monitor Moves for Cake
Layer
DOUBLE PRECISION   :: DisRestart = 10.0D-5   !If displacement size decreases to
this amount, restart it

!=====================================================================


!=====================================================================
!Energy Parameters
DOUBLE PRECISION          ::      HamSS                    =1.65D-21
!Particle-particle Hamaker constant (J)
DOUBLE PRECISION          ::      HamPS                    =1.65D-21
!Particle-wall Hamaker constant (J)
DOUBLE PRECISION          ::      SS_VDWCutoff             =      50.0D0
!Ratio center to center distance vs. particle radius
DOUBLE PRECISION          ::      C_0              299792458.0D0
!Speed of light (m/s)
DOUBLE PRECISION          ::      Eps_r            =      78.54D0
!Relative dielectric constant
DOUBLE PRECISION          ::      Electron         =      1.6D-19
!Electron charge (C)
!DOUBLE PRECISION          ::    VDWCutoff        =     0.0D0
!Minimum seperation distance to prevent VDW divergence (m)
DOUBLE PRECISION          ::      VDWCutoff        =      0.158D-9
!Minimum seperation distance to prevent VDW divergence (m)
```

```
DOUBLE PRECISION :: ElecConc = 0.001D0 !Ionic Strength (M)
DOUBLE PRECISION :: SS_EDLCutoff = 0.1D0 !Ratio center to center distance vs. particle radius
DOUBLE PRECISION :: WallZeta = -30.0D-3 !Wall zeta potential (V)
DOUBLE PRECISION :: HeightCutoff = 2.0D0 !Fraction of range where movement is not counted (considered adsorbed)
DOUBLE PRECISION :: SurfPotential = -30.0D-3 !Surface particle zeta potential (V)
INTEGER :: Surf = 2 !Number of surface particles
DOUBLE PRECISION :: ElecValence = 1.0D0 !Electrolyte charge number
DOUBLE PRECISION :: SurfValence = 1.0D0
DOUBLE PRECISION :: Valence = 1.0D0 !Particle charge number
DOUBLE PRECISION :: WallValence = 1.0D0 !Wall charge number (0.1 nm radius)
INTEGER :: EDLChoice = 0 !1 for two-range solution, any other number for single range
```

Makefile

```
now=$(shell date '+%Y%m%d_%H%M%S')
hms=$(shell date '+%H%M%S')
fortran=mpif90
ifortopt=
editor=nano
surf=NP_2100_Proc_14_Re_1000_Surf_1
srcfile=HS15
subfile=$(srcfile)sub
txtfile=./data/$(now)/JC_$(now).txt
rundir =$(shell pwd)



all:	check $(srcfile).o $(subfile).o
#	rm $(srcfile).x
	@echo "
	@echo '	~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~'
	@echo '	~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~'
	@echo '	Message: $(srcfile).x will be created.'
	@echo '	~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~'
	@echo '	~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~'
	@echo "
	$(fortran) -shared-intel $(srcfile).o $(subfile).o -
L/opt/apps/intel/11.1.064/mkl/lib/em64t -lmkl_intel_lp64 -lmkl_sequential -lmkl_core -
lguide -lpthread -o $(srcfile).x
#	$(forOAtran)   $(srcfile).o $(subfile).o -o $(srcfile).x
	@echo "
	rm $(srcfile).o $(subfile).o

$(subfile).o:
	$(fortran)  -c $(subfile).f90

$(srcfile).o:
	$(fortran)  -c $(srcfile).f90


src:	clean.src.o
	$(fortran)  -c $(ifortopt) $(srcfile).f90
	@rm $(srcfile).o
	@echo

sub:	clean.sub.o
	$(fortran)  -c $(ifortopt) $(subfile).f90
	rm $(subfile).o
```

```
	@echo

check: clean.o
	@echo '
	************************************************************'
	@echo '		Message: $(srcfile).f90 and $(subfile).f90 will be checked.'
	@echo '
	************************************************************'
	$(fortran)  -c $(ifortopt) $(srcfile).f90
	$(fortran)  -c $(ifortopt) $(subfile).f90
	rm $(srcfile).o $(subfile).o
	@echo

run:
	mkdir ./data/$(surf)
	cp /users/pmboyle/DoubleFix/vmdperl8.pl /users/pmboyle/DoubleFix/Makefile
/users/pmboyle/DoubleFix/*.f90  /users/pmboyle/DoubleFix/*.h
/users/pmboyle/DoubleFix/*.m /users/pmboyle/DoubleFix/setfiles ./data/$(surf)
	cp /users/pmboyle/DoubleFix/HS15.x ./data/$(surf)/HS15.x.$(surf)
	cp /users/pmboyle/DoubleFix/*.pbs ./data/$(surf)
	cd data/$(surf)
	qsub ./myjob.pbs

clean:  clean.o clean.x
	@echo

clean.x:
	@echo '		========================================='
	@echo '		Message: $(srcfile).x will be deleted.'
	@echo '		========================================='
	if test -e "$(srcfile).x"; then \
	(rm $(srcfile).x*) \
	fi;

clean.o:	clean.src.o clean.sub.o

clean.src.o:
	@echo '		========================================='
	@echo '		Message: $(srcfile).o will be deleted.'
	@echo '		========================================='
	if test -e "$(srcfile).o"; then \
	(rm $(srcfile).o )\
	fi

clean.sub.o:
	@echo '		========================================='
```

```
	@echo '		Message: $(subfile).o will be deleted.'
	@echo '		================================================='
	if test -e "$(subfile).o"; then \
	(rm $(subfile).o) \
	fi;


init:
#	if test -d "./Jobs"; then \
#	( echo ;  echo "Job directory exist." ;  echo )\
#	else \
#	( mkdir ./Jobs ; echo; echo "Job directory is created."; echo )\
#	fi;
	if test -d "./data"; then \
	( echo ;  echo "Data directory exist." ;  echo )\
	else \
	( mkdir ./data ; echo; echo "data directory is created."; echo )\
	fi;
```

myjob.pbs

```
#!/bin/bash
#PBS -N DoubleFix
#PBS -q compute
#PBS -l nodes=7:ppn=2,walltime=8:00:00
#PBS -S /bin/bash
#PBS -M pmboyle@rice.edu
##PBS -W x=NACCESSPOLICY:SINGLEJOB
#PBS -V
#PBS -m abe
#PBS -o /users/pmboyle/DoubleFix/data/NP_2100_Proc_14_Re_1000_Surf_1
#PBS -e /users/pmboyle/DoubleFix/data/NP_2100_Proc_14_Re_1000_Surf_1

######
echo "My job ran on:"
cat $PBS_NODEFILE
cd $PBS_O_WORKDIR
now=`date '+%Y%m%d_%H%M%S'`
hms=`date '+%H%M%S'`
echo $now

cd $HOME/DoubleFix/data/NP_2100_Proc_14_Re_1000_Surf_1
mpiexec $XD1LAUNCHER HS15.x.NP_2100_Proc_14_Re_1000_Surf_1 $now
```

APPENDIX C: Plotting Routines

HSPlot2.m

MATLAB script to plot concentrations generated in the main program.

```
figure(2) ;
load PlotDensityZ.dat
D  = PlotDensityZ;
plot(D(:,2),D(:,1),'go-',D(:,3),D(:,1),'bo-',D(:,4),D(:,1),'r-x')
volfracInitial=mean(D(:,2))
volfracFinal=mean(D(:,3))
volfracAvg  =mean(D(:,4))
axis([0 max((D(:,3)))*1.5 -1.5  1.5])
grid on
legend('Initial Concentration','Final Concentration','Avaerage Concentration')
xlabel('Volume Fraction')
ylabel('Normalized Channel Height')
```

DerVel.m

MATLAB script to plot velocity, shear, and the first derivative of shear profiles generated from coefficients output by the main program.

```
load VelCoeff.dat;
BoxHalf=112.8394*5E-6/2;
xx=linspace(-1,1,1000);
xx=xx';
Vel=zeros(1000,1);
for i=0:49
   T(:,i+1)=cos(i*acos(xx));
   Vel=Vel+VelCoeff(i+1)*T(:,i+1);
end
T1(:,1)=zeros(1000,1);  T1(:,2)=ones(1000,1);
for i=2:49
   T1(:,i+1) = 2*xx.*T1(:,i) + 2*T(:,i) - T1(:,i-1);
end
Shear=zeros(1000,1);
for i=1:50
   Shear=Shear+VelCoeff(i)*T1(:,i)/BoxHalf;
end
T2(:,1:2)=zeros(1000,2);    T1(:,3)=4*ones(1000,1);
for i=2:49
   T2(:,i+1) = 2*xx.*T2(:,i) + 4*T1(:,i) - T2(:,i-1);
end
DerShear=zeros(1000,1);
for i=1:50
   DerShear=DerShear+VelCoeff(i)*T2(:,i)/BoxHalf^2;
end
figure(1)
plot(Vel,xx)
xlabel('Velocity (m/s)')
ylabel('Normalized Channel Height')
title('Channel Velocity Profile')
print('Velocity.eps','-deps')
figure(2)
plot(Shear,xx)
xlabel('Shear Rate (1/s)')
ylabel('Normalized Channel Height')
title('Channel Shear Rate Profile')
print('Shear.eps','-deps')
figure(3)
plot(DerShear,xx)
```

ConcPlot.m

MATLAB script to plot concentration generated from coefficients output by the main program.

```
j=0;
load ConcCoeff.dat
BoxHalf=(((4/3*pi*2100)/0.1)/21)^(1/3)*7*3.13E-9/2;
[m,n]=size(ConcCoeff);
x=0:49;
x=x';
x=cos(x*pi/(49));
%size(x)
xx=zeros(490,1);
for i=0:48
   xx(i*10+1:i*10+10,1)=(linspace(x(i+1),x(i+2),10))';
end
xx=xx*BoxHalf;
ConcCoeffOld=zeros(4*49,1);
while j==0
   load ConcCoeff.dat
   pause(3)
   if ConcCoeff!=ConcCoeffOld
      conc=zeros(490,1);
      for i=0:48

conc(i*10+1:i*10+10,1)=ConcCoeff(4*i+1)*ones(10,1)+ConcCoeff(4*i+2)*xx(i*10+1:i
*10+10,1)+ConcCoeff(4*i+3)*xx(i*10+1:i*10+10,1).^2+ConcCoeff(4*i+4)*xx(i*10+1:i
*10+10,1).^3;
      end
      plot(conc,xx)
      xlabel('Volume Fraction')
      ylabel('Channel Position (m)')
      title('Concentration')
      ConcCoeffOld=ConcCoeff;
      %conc(490,1)
      %j=1;
   end
end
```

WallFraction.m

MATLAB script to plot time required for particles to reach the wall.

```
clear all
format long
load Time.dat
Radius=3.13E-9;
Area=375.8408184542459*1127.522455362738*Radius^2*1E4; %in cm^2
ParticleMass=(4/3*Radius^3*pi*1.0E3)/0.729;
%HeightCutoff=2630.885729179722/50;
HeightCutoff=2;
height=Time(:,1);
t=Time(:,2);
MaxTime=max(t);
Intervals=1000;
TimeInterval=MaxTime/Intervals;
Particles=size(height);
Divided=zeros(Intervals+1,2);
for i=0:Intervals
   Divided(i+1,1)=i*TimeInterval;
   p=0;
   for j=1:Particles(1)
      if (height(j)<=HeightCutoff)
         p=p+1;
         height2(p,:)=Time(j,:);
      end
      if ((height(j)<=HeightCutoff)&(t(j)<=Divided(i+1,1)))
         Divided(i+1,2)=Divided(i+1,2)+1;
      end
   end
end
height2;
figure(1)
plot(Divided(:,1),Divided(:,2)*ParticleMass*1E12/Area)
xlabel('Time (seconds)')
ylabel('Particle Mass in Region (ng/cm^2)')
title('Particles Mass in Bottom 2% of Cell')
figure(2)
Divided(:,2)=Divided(:,2)./Particles(1);
plot(Divided(:,1),Divided(:,2))
xlabel('Time (seconds)')
ylabel('Fraction of Particles in Region')
title('Particles in Bottom 2% of Cell')
```